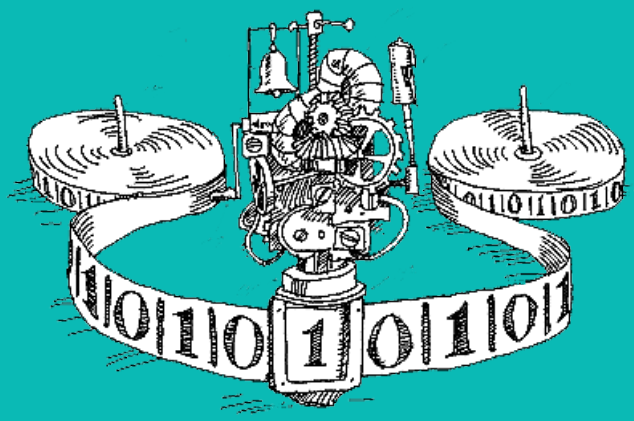
Università degli Studi di Napoli Federico II
Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea in Informatica

Appunti del corso 2014/2015 di

Elementi di INFORMATICA TEORICA

dello studente Luigi L. L. Starace



Indice

Capitolo 1

Introduzione

Questo documento è stato compilato l'ultima volta il **9 aprile 2015**. Visita il thread indicato nella sezione sottostante per sapere se sono disponibili versioni più recenti.

1.1 Informazioni su questo documento

Questo documento è da considerarsi una ricopiatura con piccole rielaborazioni ed integrazioni degli appunti presi durante le lezioni frontali del Prof. Tamburrini (corso del 2014/2015). Non ha la pretesa di essere privo di errori o completo, ma può considerarsi una buona alternativa agli altri appunti che circolano tra gli studenti del corso, più volte deprecati dallo stesso docente e sconsigliati perchè seguono un'impostazione diversa da quella che ha dato al corso negli ultimi anni.

Con la speranza che possa essere utile a molti vi ricordo che, per segnalare eventuali errori e/o trovare l'ultima versione del documento, si può fare riferimento al thread dedicato sul forum degli studenti di informatica qui: http://informatica-unina.com/forum/viewtopic.php?f=70&t=758.

1.1.1 Collegamenti cliccabili (quasi) ovunque!

Nel caso vi fosse sfuggito, vi segnalo che tutti gli argomenti citati nell'indice, nella lista di teoremi, nella lista di definizioni nonchè i riferimenti a capitoli/sezioni/teoremi/definizioni specifici e/o indirizzi web in cui vi imbatterete durante la lettura sono cliccabili e vi porteranno direttamente alla risorsa in questione!

1.1.2 Ringraziamenti e note varie

Questo documento è stato scritto in LaTeX usando **TeXworks** con l'aiuto di ottimi tool quali il **Finite State Machine Designer** di Evan Wallace (http://madebyevan.com/fsm/) e, per alcune tabelle, di **Tables Generator** (http://www.tablesgenerator.com/).
L'illustrazione della macchina di Turing presente in copertina è di Tom Dunne, pubblicata in "American Scientist".
Le citazioni presenti all'inizio di alcuni capitoli non sono realmente attribuibili ai personaggi indicati sotto di esse, qualora ciò non fosse già chiaro ☺!

1.2 Informazioni sul corso di EDIT 2014/2015

Riporto le seguenti informazioni dalla pagina del Prof. Tamburrini su docenti.unina:
www.docenti.unina.it/GUGLIELMO.TAMBURRINI

1.2.1 Contenuti

Automa finiti e macchine sequenziali. Automi non deterministici. Automa ridotto. Linguaggi regolari. Espressioni regolari. Pumping lemma per i linguaggi regolari. Grammatiche e linguaggi indipendenti dal contesto. Forme normali di Chomski e di Greibach. Automi a pila. Corrispondenza tra macchine e grammatiche. Pumping lemma per i linguaggi indipendenti dal contesto. La gerarchia di Chomsky. I concetti di algoritmo, funzione calcolabile e parzialmente calcolabile. Funzioni primitive ricorsive. La minimalizzazione. Funzioni parziali ricorsive. Numerazioni di Goedel. Macchina universale. Predicati e funzioni per la forma normale. Teorema della forma normale. Tesi di Church - Turing. Problemi di decisione. Indecidibilità. Insiemi ricorsivi e ricorsivamente numerabili. Complessità computazionale: nozioni di base.

1.2.2 Modalità di accertamento del profitto

Lo studente dovrà sostenere due prove: 1. Prova scritta in aula, volta ad accertare la capacità di risoluzione di problemi elementari di informatica teorica 2. Colloquio orale su nozioni e risultati di base dell'informatica teorica.

1.2.3 Libri di testo consigliati

Libri di testo (presenti presso la Biblioteca di Scienze a MSA):

1. M. Davis, R. Sigal, E. J. Weyuker: Computability, Complexity, and Languages (2nd ed.), Academic Press, San Diego and London.
2. M. Sipser: Introduction to the Theory of Computation (2nd ed., International edition), Thomson, London.

1.2.4 Programma di studio dai libri di testo

Dal Davis: Elementi di teoria della calcolabilità, di teoria degli automi e di teoria dei linguaggi formali. Specificamente:

Capitolo 9 Linguaggi regolari: da p. 237 a p. 262, esclusa la dimostrazione dei lemmi 1, 2 e 3 alle pp. 245-46.

Capitoli 1-4 Calcolabilità (e nozioni preliminari): da p. 1 a p.84.

Dal Sipser: Grammatiche indiendenti dal contesto e automi a pila. Specificamente:

Capitolo 2 Grammatiche indipendenti dal contesto: da p. 101 a p. 136, esclusa la dimostrazione del lemma 2.27 da p. 121 a 124.

Parte I

Calcolabilità

Capitolo 2

S-Programmi e funzioni calcolabili

"*Begin at the beginning,*" the S-Program said, very gravely, "*and go on till you come to the end: then stop.*"

Martin Davis

2.1 $\mathscr{S}$: Un linguaggio di programmazione universale

Iniziamo descrivendo il linguaggio di programmazione $\mathscr{S}$. In questo linguaggio, le *variabili in ingresso* verranno indicate con:

$$X_1\ X_2\ X_3\ \cdots\ X_n\ \cdots$$

La *variabile in uscita* è Y. Le *variabili locali* avranno nomi:

$$Z_1\ Z_2\ Z_3\ \cdots\ Z_n\ \cdots$$

Talavolta il pedice 1 viene semplicemente omesso, quindi in tal caso si intenderà $X = X_1$. Il linguaggio $\mathscr{S}$ ha tre istruzioni, descritte nella seguente tabella:

Istruzione	**Significato**
$V \leftarrow V + 1$	Incrementa di 1 il valore della variabile V.
$V \leftarrow V - 1$	Se il valore della variabile V è 0, non fa nulla; altrimenti decrementa di 1 il valore della variabile V.
$IF\ V \neq 0\ GOTO\ L$	Se il valore di V è diverso da 0, esegui l'istruzione con etichetta L come prossima; altrimenti procedi alla prossima istruzione nella lista.

Le *etichette* si indicano con:

$$A_1\ B_1\ C_1\ D_1\ E_1\ A_2\ B_2\ C_2\ D_2\ E_2\ A_3\ \cdots$$

Un'istruzione può essere o meno etichettata. Un'istruzione etichettata ha questa forma:

$$[\mathrm{A}]\ \ V \leftarrow V + 1$$

Per convenzione, le variabili locali e la variabile d'uscita hanno valore iniziale uguale a 0.
Definiamo un **$\mathscr{S}$-Programma** come una successione (lista) finita di istruzioni di incremento, decremento, salto condizionato.

2.2 Alcuni esempi di $\mathscr{S}$-Programmi

2.2.1 Primo esempio

```
1  [A]   X ← X − 1
2        Y ← Y + 1
3        IF X ≠ 0 GOTO A
```

Il programma sopra calcola $f(x) = \begin{cases} x & \text{se } x > 0 \\ 1 & \text{se } x = 0 \end{cases}$

2.2.2 La macro GOTO L

```
1  Z ← Z + 1
2  IF Z ≠ 0 GOTO L
```

Dal momento che il controllo alla riga 2 del programma sarà sempre vero, possiamo immaginare di condensare il programma nella macro **GOTO L**, che useremo in futuro per brevità e chiarezza.

2.2.3 Un programma che calcoli la funzione identità

Il seguente programma calcola $f(x) = x$:

```
1        IF X ≠ 0 GOTO A
2        GOTO E
3  [A]   X ← X − 1
4        Y ← Y + 1
5        IF X ≠ 0 GOTO A
```

Al rigo due ho usato la macro GOTO definita nella sotto-sezione precedente.

2.2.4 La macro $V \leftarrow 0$ - Azzeramento di una variabile

Consideriamo il seguente $\mathscr{S}$-Programma che azzera la variabile X:

```
1  [A]   X ← X − 1
2        IF X ≠ 0 GOTO A
```

2.2.5 La macro di assegnazione $V \leftarrow V'$

```
1  [A]   IF X ≠ 0 GOTO B
2        GOTO C
3  [B]   X ← X − 1
4        Y ← Y + 1
5        Z ← Z + 1
6        GOTO A
7  [C]   IF Z ≠ 0 GOTO D
8        GOTO E
9  [D]   Z ← Z − 1
10       X ← X + 1
11       GOTO C
```

Questo programma copia in Y il valore di X senza azzerare X. Per farlo copia dapprima il valore di X in Y e Z (azzerando X), quindi usa Z per ricopiare il valore originale in X. D'ora in avanti consideriamo questo programma condensato nella macro $V \leftarrow V'$.

2.2.6 Somma di due variabili

```
1        Y ← X1
2        Z1 ← X2
3  [B]   IF  Z1 ≠ 0 GOTO A
4        GOTO E
5  [A]   Z1 ← Z1 − 1
6        Y ← Y + 1
7        GOTO B
```

2.2.7 Prodotto di due variabili

```
1        Z2 ← X2
2  [B]   IF  Z2 ≠ 0 GOTO A
3        GOTO E
4  [A]   Z2 ← Z2 − 1
5        Z1 ← Y + X1
6        Y ← Z1
7        GOTO B
```

Oltre alla macro di salto incondizionato, ho utilizzato anche quella di somma descritta nella sotto-sezione precedente.

2.3 Funzioni totali e funzioni parziali

2.3.1 Un problema di analisi

Consideriamo il seguente $\mathscr{S}$-Programma:

```
1        Y ← X1
2        Z ← X2
3  [C]   IF  Z ≠ 0 GOTO A
4        GOTO E
5  [A]   IF  Y ≠ 0 GOTO B
6        GOTO A
7  [B]   Y ← Y − 1
8        Z ← Z − 1
9        GOTO C
```

Il programma sopra riportato calcola la seguente funzione: $f(x_1, x_2) = \begin{cases} x_1 - x_2 & \text{se } x_1 \geqslant x_2 \\ \uparrow \ (diverge) & \text{altrimenti} \end{cases}$
Infatti, se le variabili in input sono tali che $x_1 < x_2$ il programma resta "in loop" tra le righe 5-6.

2.3.2 Dominio di definizione di una funzione

Definizione 2.1 (Dominio di definizione)**.** Il **dominio di definizione di una funzione** è l'insieme di tutti gli argomenti a : $\exists$ un elemento b del codominio della funzione tale che $f(a) = b$.

Alla luce di quanto detto, il dominio di definizione della funzione calcolata dal programma nella sotto-sezione precedente è $H = \{(a, b) \in \mathbb{N}^2 \ : \ a \geqslant b\} \subset \mathbb{N}^2$.

2.3.3 Funzioni totali e funzioni parziali

Una **funzione totale** è una funzione il cui dominio di definizione è il dominio di tutti gli argomenti, cioè è una funzione che converge per qualsiasi argomento in input. Una **funzione parziale** non totale, invece, è una funzione che non converge per qualsiasi input, cioè che diverge per almeno un input.
Più precisamente:

Definizione 2.2 (Funzione Parziale)**.** Sia $f : \mathbb{N}^m \to \mathbb{N}$. f è una **funzione parziale** su $\mathbb{N}^m$ se il suo dominio di definizione D è un sottoinsieme di $\mathbb{N}^m$, cioè $D \subseteq \mathbb{N}^m$. Si noti che non è richiesta l'inclusione stretta, quindi, ad esempio, la funzione somma calcolata dal programma in 2.2.6 è una funzione parziale che è anche totale. Ad un $\mathscr{S}$-Programma si può sempre associare una funzione parziale.

2.4 Sintassi

Descriviamo in maniera più precisa il linguaggio $\mathscr{S}$ dando le seguenti definizioni:

2.4.1 S-Asserzione

Definizione 2.3 (S-Asserzione)**.** Una **$\mathscr{S}$-Asserzione** (*statement*) ha una delle seguenti forme:
$V \leftarrow V + 1$
$V \leftarrow V - 1$
$V \leftarrow V$
IF $V \neq 0$ GOTO L

Con V variabile qualsiasi e L etichetta qualsiasi.

2.4.2 S-Istruzione

Definizione 2.4 (S-Istruzione)**.** Una **$\mathscr{S}$-Istruzione** è una $\mathscr{S}$-Asserzione oppure una $\mathscr{S}$-Asserzione preceduta dalla stringa [L], con L etichetta.

2.4.3 S-Programma

Definizione 2.5 (S-Programma)**.** Un **$\mathscr{S}$-Programma** è una sequenza finita di **$\mathscr{S}$-Istruzioni**. La **lunghezza** di un $\mathscr{S}$-Programma $\mathscr{P}$ è la lunghezza di tale sequenza finita di istruzioni.

Il programma di lunghezza 0 è il **programma vuoto** che calcola la funzione nulla $f(x) = 0$.

2.4.4 Stato di un S-Programma

Definizione 2.6 (Stato di un S-Programma)**.** Lo **stato** di un $\mathscr{S}$-Programma $\mathscr{P}$ è una lista di equazioni della forma $V = m,\ m \in \mathbb{N}$ tale che:

(i) Vi è un'equazione per ogni variabile di $\mathscr{P}$.

(ii) Non vi sono due equazioni che iniziano con la stessa variabile.

2.4.5 Istantanea di un S-Programma

Definizione 2.7 (Istantanea di un S-Programma)**.** Un'**istantanea** (*snapshot*) di un programma $\mathscr{P}$ che abbia lunghezza n è una *coppia ordinata* (i, σ), dove

- i dev'essere un numero naturale compreso tra 1 e n (*prossima istruzione*).
- σ è uno stato di $\mathscr{P}$.

2.4.6 Istantanea terminale

Definizione 2.8 (Istantanea Teminale)**.** Un'istantanea si dice **terminale** (*che non ha successori*) se $i = n + 1$, dove n è la lunghezza del programma. Quando si incontra un **GOTO L**, con L etichetta non definita, si pone $i = n + 1$.

2.4.7 Istantanea successore

Definizione 2.9 (Istantanea successore)**.** Se un'istantanea (i, σ) non è terminale, è possibile definire l'istantanea **successore** di (i, σ) in questo modo:

Caso 1 La i-esima istruzione è un incremento $V \leftarrow V + 1$. σ conterrà l'equazione $V = m$. L'istantanea successore avrà forma $(i + 1, \tau)$, dove lo stato τ differisce dallo stato σ soltanto per l'equazione $V = m + 1$ che sostituisce $V = m$.

Caso 2 La i-esima istruzione è un decremento $V \leftarrow V - 1$. σ conterrà l'equazione $V = m$. L'istantanea successore avrà forma $(i + 1, \tau)$, dove lo stato τ differisce dallo stato σ soltanto per l'equazione $V = m - 1$ che sostituisce $V = m$.

Caso 3 La i-esima istruzione è un'istruzione pigra $V \leftarrow V$. L'istantanea successore avrà forma $(i+1, \sigma)$.

Caso 4 La i-esima istruzione ha forma IF $V \neq 0$ GOTO L. Allora lo stato dell'istantanea successore $\tau = \sigma$. Quindi si distinguono due casi:

4.a Se σ contiene l'equazione $V = 0$, l'istantanea successore sarà $(i + 1, \tau)$.

4.b Se σ non contiene l'equazione $V = 0$, l'istantanea successore sarà (j, τ) e si distinguono a loro volta due casi:

4.b.1 Se $\mathscr{P}$ contiene almeno un'istruzione etichettata L, allora j sarà il più piccolo numero naturale tale che la j-esima istruzione è etichettata [L]

4.b.2 $j = n + 1$ altrimenti.

2.5 Funzioni calcolabili

2.5.1 Calcolo terminante e non

Un programma nel linguaggio $\mathscr{S}$ termina quando raggiunge un'istantanea terminale. Un calcolo di un programma $\mathscr{P}$ nel linguaggio $\mathscr{S}$ può essere:

Terminante se è descrivibile tramite una successione finita di istantanee $S_1, \cdots, S_k$ tale che $S_k = (n + 1, \gamma)$ (cioè S_k terminale) e S_{i+1} è il successore di S_i per ogni $i : 1 \leqslant i \leqslant k - 1$.

Non terminante se è descrivibile tramite una successione infinita di istantanee $S_1, \cdots$ tale che S_{i+1} è il successore di S_i per ogni $i \geqslant 1$.

2.5.2 Istantanea iniziale

Siano dati un programma $\mathscr{P}$, dei numeri naturali $r_1, \cdots, r_m : m \geqslant 1$.
Definiamo lo **stato iniziale** σ di $\mathscr{P}$ su $r_1, \cdots, r_m$ in questo modo:

$$\sigma = \begin{cases} X_1 = r_1 \\ X_2 = r_2 \\ \vdots \\ X_m = r_m \\ Y = 0 \\ V = 0 \text{ per ogni altra variabile V di } \mathscr{P} \end{cases}$$

Definiamo l'**istantanea iniziale** come $(1, \sigma)$. Possono quindi verificarsi due casi:

1) Vi è un calcolo terminante $S_1, \cdots, S_k$ di $\mathscr{P}$ che parte dalla istantanea iniziale. In tal caso denotiamo con
$\psi^m_{\mathscr{P}}(r_1, \cdots, r_m)$ il valore di Y nell'istantanea terminale S_k.

2) Non vi è un calcolo terminante che parte dall'istantanea iniziale $(1, \sigma)$. In tal caso
$\psi^m_{\mathscr{P}}(r_1, \cdots, r_m) = \uparrow$ (diverge, è indefinita).

2.5.3 Funzioni parzialmente calcolabili

Definizione 2.10 (Funzione parzialmente calcolabile)**.** Una funzione parziale f di n argomenti si dice **parzialmente calcolabile** se esiste un $\mathscr{S}$-Programma $\mathscr{P}$ che la calcola, cioè tale che per ogni n-pla di argomenti $r_1, \cdots, r_n$ si ha

$$f(r_1, \cdots, r_n) = \psi^n_{\mathscr{P}}(r_1, \cdots, r_n)$$

Indichiamo con $\psi^n_{\mathscr{P}}(r_1, \cdots, r_n)$ il valore della variabile Y nell'istantanea terminale della computazione del programma $\mathscr{P}$ con gli n valori in input $r_1, \cdots, r_n$.

2.5.4 Funzioni calcolabili

Definizione 2.11 (Funzione calcolabile)**.** Una funzione parzialmente calcolabile e totale si dice **calcolabile**. Esiste un programma che la calcola e tale programma termina per ogni input.

Alla luce delle due definizioni precedenti, possiamo classificare alcune delle funzioni descritte finora:

Funzione	**Parzialmente calcolabile**	**Calcolabile**
$x + y$	Sì	Sì
$x \cdot y$	Sì	Sì
$x - y$	Sì	No

2.6 Altre macro

Introduciamo nuove macro:

2.6.1 $Z \leftarrow X_1 + X_2$

Per l'espansione di questa macro si veda il programma nella sezione 2.2.6.

2.6.2 $Z \leftarrow X_1 - X_2$

Per l'espanzione di questa macro si veda il programma nella sezione 2.3.1. Si ricordi che, se $X_2 > X_1$ la macro non terminerà e non verranno eseguite le istruzioni successive.

2.6.3 $Z \leftarrow f(w_1, \cdots, w_n)$ con f parzialmente calcolabile

Più in generale possiamo immaginare la macro $Z \leftarrow f(w_1, \cdots, w_n)$ con f parzialmente calcolabile (quindi esiste un programma che la calcola). Facendo semplicemente attenzione a mantenere separate le variabili e le etichette del programma "chiamante" e del programma che calcola f, cosa non difficile dal momento che è possibile definite infinite variabili e infinite etichette.

2.6.4 Predicati

Definizione 2.12 (Predicato)**.** Un predicato è una funzione totale $\mathcal{P}(x_1, \cdots, x_n) : \mathbb{N}^n \to \{0 \text{ (falso)}, 1 \text{ (vero)}\}$. Se esiste un programma $\mathscr{P}$ che calcola un predicato $\mathcal{P}$, questo programma terminerà per ogni input (sarà **calcolabile**).

Possiamo quindi definire la macro

$$\text{IF } \mathcal{P}(x_1, \cdots, x_n) \text{ GOTO L}$$

la cui espansione è banalmente:

```
1    Z ← P(x_1, ..., x_n)
2    IF  Z ≠ 0 GOTO L
```

Capitolo 3

Funzioni primitive ricorsive

Vogliamo essere in grado di combinare tra loro funzioni calcolabili in modo tale che la funzione ottenuta sia ancora calcolabile.

3.1 Schema di composizione funzionale

Definizione 3.1 (Schema di composizione funzionale)**.** Sia f una funzione di k variabili e siano $g_1, \cdots, g_k$ funzioni di n variabili. Sia

$$h(x_1, \cdots, x_n) = f(g_1(x_1, \cdots, x_n), \cdots, g_k(x_1, \cdots, x_n))$$

Allora h si dice ottenuta da f e $g_1, \cdots, g_k$ tramite **composizione funzionale**.

Teorema 3.1 (Funzioni ottenute per composizione da funzioni calcolabili sono calcolabili)**.** *La funzione h ottenuta per composizione dalle funzioni* (parzialmente) *calcolabili $f, g_1, \cdots, g_k$ è* (parzialmente) *calcolabile.*

Dimostrazione. Esibisco un programma in $\mathscr{S}$ che calcola h:

```
1   Z_1 ← g_1(X_1, ··· , X_n)
2   ···
3   Z_k ← g_k(X_1, ··· , X_n)
4   Y ← f(Z_1, ··· , Z_k)
```

Banalmente, se $f, g_1, \cdots, g_k$ sono totali, anche h lo sarà. □

3.2 Schema di ricorsione 1

Definizione 3.2 (Schema di ricorsione 1)**.** Sia k un generico numero naturale e sia

$$\begin{cases} h(0) & = k, \\ h(t+1) & = g(t, h(t)), \end{cases}$$

dove g è una funzione totale di due variabili. Allora h si dice ottenuta da g per ricorsione primitiva (o semplicemente ricorsione).

Teorema 3.2 (Funzioni ottenute per ricorsione 1 da funzioni calcolabili sono calcolabili)**.** *Se h è ottenuta dalla funzione calcolabile g secondo lo schema definito in 3.2, allora h è calcolabile.*

Dimostrazione. Innanzitutto notiamo che la funzione costante $h(x) = k$ è calcolabile, in quanto lo S-Programma composto da k incrementi della variabile Y la calcola (ed è chiaramente totale). Possiamo utilizzare quindi la macro $V \leftarrow k$. Esibiamo un S-Programma che calcola h:

```
1        Y ← k
2  [A]   IF X = 0 GOTO E
3        Y ← g(Z, Y)
4        Z ← Z + 1
5        X ← X − 1
6        GOTO A
```

Se, come da ipotesi, g è calcolabile, allora anche h ottenuta per ricorsione primitiva da g sarà calcolabile. □

Nota. Poichè si è fatto uso della macro $X = 0$, per completezza se ne fornisce l'espansione:

```
1        IF X ≠ 0 GOTO E
2        Y ← Y + 1
```

3.3 Schema di ricorsione 2

Definizione 3.3 (Schema di ricorsione 2)**.** Sia h definita in questo modo:

$$\begin{cases} h(x_1, \cdots, x_n, 0) & = f(x_1, \cdots, x_n) \\ h(x_1, \cdots, x_n, t+1) & = g(t, h(x_1, \cdots, x_n, t), x_1, \cdots, x_n) \end{cases}$$

La funzione h di $n+1$ variabili è ottenuta per *ricorsione primitiva* (o semplicemente *ricorsione*) dalle funzioni totali f di n variabili e g di $n+2$ variabili.

Teorema 3.3 (Funzioni ottenute per ricorsione 2 da funzioni calcolabili sono calcolabili)**.** *Se h è ottenuta da f e g come descritto nella definizione 3.3, allora h è calcolabile.*

Dimostrazione. Esibisco una versione modificata del programma usato in 3.2:

```
1        Y ← f(X1, ··· , Xn)
2  [A]   IF Xn+1 = 0 GOTO E
3        Y ← g(Z, Y, X1, ··· , Xn)
4        Z ← Z + 1
5        Xn+1 ← Xn+1 − 1
6        GOTO A
```

□

3.4 Funzioni primitive ricorsive

Innanzitutto notiamo che, per i teoremi 3.1, 3.2, 3.3 che abbiamo precedentemente dimostrato, vale il seguente:

Corollario. *Le funzioni calcolabili sono chiuse sotto gli schemi di composizione (3.1) e ricorsione primitiva (3.2 e 3.3).*

3.4.1 Funzioni di base

Definiamo le sequenti funzioni di base:

$$S(x) = x + 1$$
$$n(x) = 0$$
$$u_i^n(x_1, \cdots, x_n) = x_i, \quad 1 \leqslant i \leqslant n$$

Tali funzioni sono tutte **totali** e **calcolabili**, essendo banalmente calcolate dai sequenti S-Programmi:

(a) $S(x)$:

```
1    Y ← Y + 1
```

(si ricorda che la somma è calcolabile).

(b) $n(x)$: programma vuoto (definito in 2.4.3).

(c) $u_i^n(x_1, \cdots, x_n) = x_i$:

```
1    Y ← X_i
```

Definizione 3.4 (Funzione primitiva ricorsiva)**.** Una funzione è primitiva ricorsiva se è una delle funzioni di base oppure se può essere ottenuta da esse applicando un numero finito di volte gli schemi di composizione e ricorsione primitiva.

Tutte le funzioni primitive ricorsive sono calcolabili, ma non tutte le funzioni calcolabili sono primitive ricorsive. Ad esempio la **funzione di Ackermann**, di seguito riportata, è calcolabile ma non è primitiva ricorsiva

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Definizione 3.5 (Insieme numerabile)**.** Un insieme si dice numerabile se è finito oppure se può essere messo in corrispondenza biunivoca con l'insieme $\mathbb{N}$ dei numeri naturali.

Teorema 3.4 (Le funzioni totali $f : \mathbb{N} \to \mathbb{N}$ non sono enumerabili.)**.** *La classe delle funzioni unarie su* $\mathbb{N}$ *totali non è enumerabile.*

Dimostrazione. Dimostriamo la tesi per diagonalizzazione. Supponiamo per assurdo che esista una numerazione $f_1, f_2, \cdot$. Consideriamo quindi la funzione unaria $g(x) = f_x(x) + 1$. La funzione g è chiaramente totale, in quanto primitiva ricorsiva perchè ottenuta per composizione di f (per ipotesi totale) con la funzione di base $S(x)$. Poichè g è totale e unaria, allora esiste un indice i tale che $g(x) = f_i(x)$, per ogni possibile x. Ma, per definizione stessa di g, per $x = i$ $g(i) = f_i(i) + 1$. Non è possibile fornire un'enumerazione che non cada in questa contraddizione. g viene anche detta **funzione antidiagonale**, in quanto differisce di uno da ogni punto della diagonale. La seguente tabella chiarirà la dimostrazione e l'utilizzo che si è fatto del termine diagonale:

	0	1	2	$\cdots$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$\cdots$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$\cdots$
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

□

3.5 Alcune funzioni primitive ricorsive

Procediamo enumerando alcune funzioni primitive ricorsive (quindi anche calcolabili, per il corollario).

3.5.1 Funzione somma

Per mostrare che $f(x,y) = x + y$ è primitiva ricorsiva, mostriamo che è ottenibile dalle funzioni di base utilizzando un numero finito di volte gli schemi di ricorsione primitiva e composizione. Infatti possiamo scriverla come

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

che equivale a

$$\begin{cases} f(x,0) & = u_1^1(x) \\ f(x,y+1) & = g(x, f(x,y), y) = S(u_2^3(x, f(x,y), y)) \end{cases}$$

Poichè abbiamo applicato lo schema di ricorsione 2 usando la funzione g che è primitiva ricorsiva in quanto risultato dell'applicazione dello schema di composizione alle funzioni di base e la funzione u_1^1 che è una funzione di base e quindi primitiva ricorsiva, allora la funzione somma è primitiva ricorsiva.

3.5.2 Funzione prodotto

Possiamo definire la funzione $f(x,y) = x \cdot y$ ricorsivamente come

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

che equivale a

$$\begin{cases} f(x,0) & = n(x) \\ f(x,y+1) & = g(x, f(x,y), y) = (u_2^3(x, f(x,y), y)) + x \end{cases}$$

Poichè abbiamo applicato lo schema di ricorsione 2 usando la funzione g che è primitiva ricorsiva (in quanto risultato dell'applicazione dello schema di composizione alle funzioni di base e alla funzione somma che abbiamo dimostrato nella sotto-sezione precedente essere primitiva ricorsiva) e la funzione $n(x)$ che è una funzione di base e quindi primitiva ricorsiva, allora la funzione prodotto è primitiva ricorsiva.

3.5.3 Funzione fattoriale

Diamo una definizione ricorsiva del fattoriale come

$$\begin{cases} 0! & = 1 \\ (x+1)! & = x! \cdot (x+1) \end{cases}$$

Più precisamente $f(x) = x!$ dove

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

dove $g(x_1, x_2) = u_2^2(x_1, x_2) \cdot S(u_1^2(x_1, x_2))$. g è primitiva ricorsiva in quanto risultato di composizione di funzioni di base e della funzione prodotto che abbiamo mostrato essere primitiva ricorsiva. Dunque anche il fattoriale è primitivo ricorsivo perchè abbiamo applicato lo schema di ricorsione 1 a funzioni primitive ricorsive.

3.5.4 Esponenziazione x^y

Le equazioni di ricorsione sono:

$$\begin{cases} x^0 & = 1 \\ x^{y+1} & = x^y \cdot x \end{cases}$$

Più precisamente $f(x,y) = x^y$ dove

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

dove la funzione $g(x_1, x_2, x_3) = u_2^3(x_1, x_2, x_3) \cdot u_3^3(x_1, x_2, x_3)$. Poichè g, h sono primitive ricorsive (d'ora in avanti semplicemente **PR**), allora anche f è PR.

3.5.5 Predecessore limitato $p(x)$

La funzione predecessore limitato (che in $\mathscr{S}$ corrisponde all'istruzione di decremento) è definita come segue:

$$p(x) = \begin{cases} x - 1 & \text{se } x \neq 0 \\ 0 & \text{altrimenti} \end{cases}$$

Le equazioni di ricorsione sono

$$\begin{cases} p(0) & = 0 \\ p(x+1) & = x \end{cases}$$

Più precisamente $f(x) = p(x)$ dove

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

dove $g(x_1, x_2) = u_1^2(x_1, x_2)$. Dunque, poichè ottenuta da funzioni di base applicando lo schema di ricorsione 1, $p(x)$ è primitiva ricorsiva.

3.5.6 Sottrazione limitata $x \dot{-} y$

La funzione è definita come segue

$$x \dot{-} y = \begin{cases} x - y & \text{se } x \geqslant y \\ 0 & \text{altrimenti} \end{cases}$$

Mostriamone una definizione con equazioni di ricorsione

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

che possiamo anche scrivere in questo modo

$$\begin{cases} x \dot{-} 0 & = u_1^1(x) \\ x \dot{-} (y+1) & = g(x, (x \dot{-} y), y) \end{cases}$$

con $g(x_1, x_2, x_3) = p(u_2^3(x_1, x_2, x_3))$, in modo da rendere evidente che è ricavata applicando lo schema di ricorsione 2 usando funzioni primitive ricorsive.

3.5.7 Valore assoluto di una differenza $|x - y|$

Dal momento che $|x - y| = (x \dot{-} y) + (y \dot{-} x)$, è immediato verificare che $|x - y|$ è primitiva ricorsiva, in quanto ottenuta per composizione da funzioni primitive ricorsive.

3.5.8 Funzione rilevatrice di zeri $\alpha(x)$

La funzione è definita in questo modo

$$\alpha(x) = \begin{cases} 1 & \text{se } x = 0 \\ 0 & \text{altrimenti} \end{cases}$$

ed è primitiva ricorsiva in quanto $\alpha(x) = (1 \dot{-} x)$. In alternativa giungiamo alla stessa conclusione mostrando le equazioni di ricorsione

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

con $g(x, \alpha(x)) = n(u_1^2(x_1, x_2)) = 0$.

3.6 Predicati primiti ricorsivi

La definizione di predicato è stata già data in 2.12. Continuiamo il nostro elenco di funzioni primitive ricorsive elencano alcuni predicati.

3.6.1 Uguaglianza $x = y$

Il predicato di uguaglianza corrisponde alla seguente funzione

$$d(x,y) = \begin{cases} 1 & \text{se } x = y \\ 0 & \text{altrimenti} \end{cases}$$

Tale funzione è primitiva ricorsiva, infatti corrisponde a $\alpha(|x - y|)$.

3.6.2 Predicato $x \leqslant y$

Il predicato è primitivo ricorsivo, infatti corrisponde a $\alpha(x \dot{-} y)$.

3.6.3 Chiusura dei predicati PR rispetto alle operazioni di $\neg, \wedge, \vee$

Teorema 3.5 (Chiusura dei predicati PR rispetto alle operazioni di $\neg, \wedge, \vee$)**.** *Se p e q sono predicati PR, allora anche i seguenti sono primitivi ricorsivi:*

- $\neg p$ *infatti corrisponde ad* $\alpha(p)$
- $p \wedge q$ *infatti corrisponde a* $p \cdot q$
- $p \vee q$ *infatti per la legge di De Morgan* $p \vee q = \neg((\neg p) \cdot (\neg q))$

3.6.4 Predicato $x < y$

$(x < y) = (x \leqslant y) \wedge \neg(x = y)$, dunque è primitivo ricorsivo. Ancor più semplicemente, si poteva notare che $(x < y) = \neg(x \geqslant y)$.

3.6.5 Definizione per casi

Teorema 3.6 (Definizione per casi)**.** *Sia f definita per casi come segue*

$$f(x_1, \cdots, x_n) = \begin{cases} g(x_1, \cdots, x_n) & \text{se } P(x_1, \cdots, x_n) \\ h(x_1, \cdots, x_n) & \text{se } \neg P(x_1, \cdots, x_n) \end{cases}$$

Allora, se g, h, P sono primitivi ricorsivi, f è primitiva ricorsiva.

Dimostrazione. $f(x_1, \cdots, x_n) = [g(x_1, \cdots, x_n) \cdot P(x_1, \cdots, x_n)] + [h(x_1, \cdots, x_n) \cdot \neg P(x_1, \cdots, x_n)]$.
Poichè f è ottenuta per composizione da funzioni primitive ricorsive, f è primitiva ricorsiva. □

Corollario 3.6.1 (Generalizzazione della definizione per casi)**.** *Sia f definita per casi come segue*

$$f(x_1, \cdots, x_n) = \begin{cases} g_1(x_1, \cdots, x_n) & \text{se } P_1(x_1, \cdots, x_n) \\ \vdots & \\ g_{n+1}(x_1, \cdots, x_n) & \text{se } P_{n+1}(x_1, \cdots, x_n) \\ h(x_1, \cdots, x_n) & \text{altrimenti} \end{cases}$$

con $n > 1$ e $h, g_1, \cdots, g_n, P_1, \cdots, P_n$ primitive ricorsive. Allora f è PR.

Dimostrazione. Dimostriamo la tesi usando il principio d'induzione.
Caso base: il caso per $n = 0$ è stato mostrato nel teorema 3.6.
Passo induttivo: supponiamo (ipotesi induttiva) che la tesi sia valida per tutti i numeri naturali fino ad n. Mostriamo che ciò implica che la tesi è vera anche per $n + 1$. Definiamo la funzione di servizio

$$h^\star(x_1, \cdots, x_n) = \begin{cases} g_{n+1}(x_1, \cdots, x_n) & \text{se } P_{n+1}(x_1, \cdots, x_n) \\ h(x_1, \cdots, x_n) & \text{altrimenti} \end{cases}$$

Dal teorema 3.6, $h^\star$ è PR. Consideriamo quindi la nostra funzione f come

$$f(x_1, \cdots, x_n) = \begin{cases} g_1(x_1, \cdots, x_n) & \text{se } P_1(x_1, \cdots, x_n) \\ \vdots & \\ g_n(x_1, \cdots, x_n) & \text{se } P_n(x_1, \cdots, x_n) \\ h^\star(x_1, \cdots, x_n) & \text{altrimenti} \end{cases}$$

Su f ora si applica l'ipotesi induttiva, implicando che f è PR. □

3.7 Operazioni iterate e quantificatori limitati

Teorema 3.7 (Chiusura delle funzioni PR rispetto a sommatorie e produttorie)**.** *Se $f(y, x_1, \cdots, x_n)$ è una funzione PR, allora anche le seguenti sono funzioni PR*

$$g(y, x_1, \cdots, x_n) = \sum_{t=0}^{y} f(t, x_1, \cdots, x_n)$$
$$h(y, x_1, \cdots, x_n) = \prod_{t=0}^{y} f(t, x_1, \cdots, x_n)$$

Dimostrazione. Per dimostrare la tesi diamo una definizione primitiva ricorsiva di g e h.

$$g(0, x_1, \cdots, x_n) = f(0, x_1, \cdots, x_n)$$
$$g(t+1, x_1, \cdots, x_n) = g(t, x_1, \cdots, x_n) + f(t+1, x_1, \cdots, x_n)$$

Mentre analogamente, per h

$$h(0, x_1, \cdots, x_n) = f(0, x_1, \cdots, x_n)$$
$$h(t+1, x_1, \cdots, x_n) = h(t, x_1, \cdots, x_n) \cdot f(t+1, x_1, \cdots, x_n)$$

Dal momento che somma e prodotto sono primitive ricorsive e che f è, per ipotesi, primitiva ricorsiva, allora anche g e h sono primitive ricorsive. □

Si noti che, volendo far partire la sommatoria e la produttoria da 1 anzichè da 0, sarebbe sufficiente modificare il secondo membro l'equazione del "caso base" con l'elemento neutro rispettivamente della somma è del prodotto, come mostrato nel corollario seguente.

Corollario 3.7.1. *Se $f(y, x_1, \cdots, x_n)$ è una funzione PR, allora anche le seguenti sono funzioni PR*

$$g(y, x_1, \cdots, x_n) = \sum_{t=1}^{y} f(t, x_1, \cdots, x_n)$$
$$h(y, x_1, \cdots, x_n) = \prod_{t=1}^{y} f(t, x_1, \cdots, x_n)$$

Dimostrazione. Per dimostrare la tesi diamo una definizione primitiva ricorsiva di g e h.

$$g(0, x_1, \cdots, x_n) = 0$$
$$g(t+1, x_1, \cdots, x_n) = g(t, x_1, \cdots, x_n) + f(t+1, x_1, \cdots, x_n)$$

Mentre analogamente, per h

$$h(0, x_1, \cdots, x_n) = 1$$
$$h(t+1, x_1, \cdots, x_n) = h(t, x_1, \cdots, x_n) \cdot f(t+1, x_1, \cdots, x_n)$$

□

3.7.1 Quantificatori limitati

Teorema 3.8 (Chiusura dei predicati PR rispetto ai quantificatori limitati)**.** *Sia il predicato $P(t, x_1, \cdots, x_n)$ primitivo ricorsivo. Allora anche i predicati*

$$(\forall t)_{\leqslant y} P(t, x_1, \cdots, x_n) \text{ e } (\exists t)_{\leqslant y} P(t, x_1, \cdots, x_n) \text{ sono primitivi ricorsivi.}$$

Dimostrazione. È sufficiente osservare che

$$(\forall t)_{\leqslant y} P(t, x_1, \cdots, x_n) \Leftrightarrow \left[\prod_{t=0}^{y} P(t, x_1, \cdots, x_n)\right] = 1$$

e che

$$(\exists t)_{\leqslant y} P(t, x_1, \cdots, x_n) \Leftrightarrow \left[\sum_{t=0}^{y} P(t, x_1, \cdots, x_n)\right] \neq 0.$$

Da ciò e dal teorema 3.7 segue la nostra tesi. □

3.8 Altri predicati primitivi ricorsivi

3.8.1 y è un divisore di x

$x|y \Leftrightarrow (\exists t)_{\leqslant y} \ : \ t \cdot x = y$. Poichè è quantificatore esistenziale limitato di un predicato primitivo ricorsivo, $x|y$ è primitivo ricorsivo.

3.8.2 Primo(x)

Il predicato Primo(x), che ritorna vero se e solo se x è primo, equiavale al seguente:

$$(\forall y)_{\leqslant x} \left[(y > 0) \wedge ((y = 1) \vee (y = x) \vee \neg(x|y))\right]$$

Poichè quantificatore universale limitato di un predicato PR (perchè ottenuto per composizione di predicati PR), allora Primo(x) è PR.

3.9 Minimalizzazione

Sia $P(t, x_1, \cdots, x_n)$ un predicato PR. Allora la seguente

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

è primitiva ricorsiva per il teorema 3.7 precedente. Analizziamo cosa calcola g. Sia t_0 il più piccolo valore di $t \leqslant y$ per cui P è vero. Consideriamo quindi la produttoria interna

$$\prod_{t=0}^{u} \alpha(P(t, x_1, \cdots, x_n)) = \begin{cases} 1 & \text{se } y < t_0 \\ 0 & \text{se } y \geqslant t_0 \end{cases}$$

Quindi g estrae, se esiste, il più piccolo numero per il quale P è vero.
Definiamo dunque la minimalizzazione limitata come segue

$$\min_{t \leqslant y} P(t, x_1, \cdots, x_n) = \begin{cases} g(t, x_1, \cdots, x_n) & \text{se } (\exists t)_{\leqslant y} \ : \ P(t, x_1, \cdots, x_n) \\ 0 = n(x_1) & \text{altrimenti} \end{cases}$$

Teorema 3.9 (La minimalizzazione limitata è PR)**.** *Se $P(t, x_1 m \cdots, x_n)$ è un predicato PR, allora $\min_{t \leqslant y} P(t, x_1, \cdots, x_n)$ è PR.*

Dimostrazione. $\min_{t \leqslant y}$ è definita per casi usando funzioni e predicati primitivi ricorsivi. Infatti g è stata dimostrata PR inpoche righe fa per il teorema 3.7, e il predicato è PR perchè quantificatore esistenziale limitato di predicato, per ipotesi, PR. □

Si noti che questa definizione presenta una piccola ambiguità. Nel caso in cui il risultato trovato sia 0, infatti, è necessario testare $P(0, x_1, \cdots, x_n)$ per vedere se la risposta non sia proprio 0!

3.9.1 Minimalizzazione non limitata

Se consideriamo la minimalizzazione non limitata, come ad esempio

$$\underset{z}{min}\,[y + z = x]$$

usciamo dalle funzioni primitive ricorsive (e calcolabili). Infatti, se nell'esempio proposto fosse $y > x$, allora $\underset{z}{min}$ diverge!

Teorema 3.10 (La minimalizzazione non limitata è parzialmente calcolabile)**.** *Sia $P(t, x_1, \cdots, x_n)$ un predicato calcolabile e sia*

$$g(x_1, \cdots, x_n) = \underset{t}{min}\,[P(t, x_1, \cdots, x_n)]$$

Allora g è parzialmente calcolabile.

Dimostrazione. Si consideri il seguente S-Programma che calcola g

```
1 [A]   IF  P(Z, X_1, ..., X_n)  GOTO B
2       Z ← Z + 1
3       GOTO A
4 [B]   Y ← Z
```

□

3.10 Altre funzioni primitive ricorsive

3.10.1 Funzione enumeratrice di primi

La funzione enumeratrice di primi p_n restituisce l'n-esimo numero primo. Affinchè sia **totale**, per definizione imporremo $p_0 = 0$. Prima di dimostrare che p_n è primitiva ricorsiva, dimostriamo il seguente teorema:

Teorema 3.11 (Teorema sulla densità dei numeri primi)**.** *Per ogni $n \in \mathbb{N}$ vale la seguente*

$$p_n + 1 \leqslant p_{n+1} \leqslant (p_n)! + 1$$

Dimostrazione. Si consideri il numero $(p_n)! + 1$. Esso non è divisibile per nessuno dei primi minori p_n (si avrà sempre resto 1, banalmente). Quindi possono verificarsi due casi:

Caso 1: $(p_n)! + 1$ è primo e allora, anche se fosse proprio l'(n+1)esimo primo, la disequazione varrebbe.

Caso 2: $(p_n)! + 1$ non è primo. In tal caso esisterà almeno un primo $q \;:\; p_n < q < (p_n)! + 1$ e la disequazione varrebbe ancora.

Da ciò (e dall'infinità dei numeri primi, in tal proposito si veda una delle numerose dimostrazioni, come ad esempio quella di Euclide, non riportata qui per brevità), segue il teorema. □

Quindi possiamo mostrare che la funzione enumeratrice di primi è PR. Innanzitutto definiamo la seguente funzione PR

$$h(y, z) = \underset{t \leqslant z}{min}\,[Primo(t) \wedge t > y]$$

e quindi la funzione

$$k(x) = h(x, x! + 1)$$

anch'essa PR. Ora possiamo scrivere p_n in forma ricorsiva nel seguente modo:

$$p_0 = 0$$
$$p_{n+1} = k(p_n)$$

concludendo che p_n è primitiva ricorsiva.

3.10.2 Parte intera del quoziente $\frac{x}{y}$

Diamo una definizione PR:

$$\left\lfloor \frac{x}{y} \right\rfloor \Leftrightarrow \underset{t \leqslant x}{min}\,[(t + 1) \cdot y > x]$$

3.10.3 Resto della divisione $\frac{x}{y}$

Diamo una definizione PR:

$$R\left(\frac{x}{y}\right) \Leftrightarrow x \dot{-} \left(y \cdot \left\lfloor \frac{x}{y} \right\rfloor\right)$$

Capitolo 4

Funzione angoletto e numeri di Gödel

In questo capitolo studieremo due modi utili di codificare informazioni. Il primo, la **funzione angoletto**, permetterà di codificare coppie di numeri con un unico numero, mentre il secondo, la **codifica di Gödel**, permetterà di codificare successioni di numeri.

4.1 Funzione angoletto

Definizione 4.1 (Funzione angoletto)**.** La funzione primitiva ricorsiva "angoletto" è definita in questo modo

$$\langle x, y\rangle = 2^x(2y+1) \dot{-} 1.$$

Poichè $2^x(2y+1) \neq 0$,

$$\langle x, y\rangle + 1 = 2^x(2y+1).$$

Per ogni dato z, esiste un'unica soluzione x, y all'equazione

$$\langle x, y\rangle = z,$$

e, precisamente, x è il massimo numero tale che $2^x|(z+1)$ mentre y è la soluzione di

$$2y + 1 = \frac{z+1}{2^x}.$$

Definizione 4.2 (Funzione $l(z)$ e $r(z)$)**.** Assumendo quindi $\langle x, y\rangle = z$ definiamo le funzioni "inverse" $l(z) = x$ e $r(z) = y$. Poichè si ha che $l(z) \leqslant z$ e che $r(z) \leqslant z$, possiamo scrivere

$$l(z) = \min_{x \leqslant z} \left[(\exists y)_{\leqslant z} \left(z = \langle x, y\rangle\right)\right],$$
$$r(z) = \min_{y \leqslant z} \left[(\exists x)_{\leqslant z} \left(z = \langle x, y\rangle\right)\right],$$

in modo che le funzioni l e r siano primitive ricorsive.
La definizione di l e r equivale anche alla seguente affermazione:

$$\langle x, y\rangle = z \Leftrightarrow x = l(z) \wedge y = r(z).$$

Teorema 4.1 (Riepilogo sulle proprietà delle funzioni $\langle x, y\rangle$, $l(z)$, $r(z)$)**.** *Le funzioni $\langle x, y\rangle$, $l(z)$, $r(z)$ precedentemente descritte godono delle seguenti proprietà:*

1) sono primitive ricorsive;

2) $l\left(\langle x, y\rangle\right) = x$, $r\left(\langle x, y\rangle\right) = y$;

3) $\langle l(z), r(z)\rangle = z$

4) $l(z), r(z) \leqslant z$

4.2 Numeri di Gödel

Definizione 4.3 (Numeri di Gödel)**.** Definiamo il *numero di Gödel* di una sequenza $(a_1, \cdots, a_n)$ in questo modo:

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

Poichè abbiamo già mostrato nel capitolo 3 che produttoria, esponenziazione, funzione generatrice di primi sono primitive ricorsive, allora anche $[a_1, \cdots, a_n]$ è primitiva ricorsiva.

Teorema 4.2 (Unicità della successione di un numero di Gödel)**.** *Se* $[a_1, \cdots, a_n] = [b_1, \cdots, b_n]$*, allora* $a_i = b_i, \quad \forall i \in 1, \cdots, n$

Dimostrazione. La dimostrazione è conseguenza diretta e banale del *Teorema fondamentale dell'aritmetica (TFA)*, per la cui dimostrazione si rimanda al testo di algebra. □

Notiamo anche che si ha

$$[a_1, \cdots, a_n] = [a_1, \cdots, a_n, 0] = [a_1, \cdots, a_n, 0, \cdots, 0].$$

Per convenzione, considereremo 1 come il numero di Gödel della sequenza "vuota" di lunghezza 0.

4.2.1 La funzione di proiezione $(x)_i$

Definizione 4.4 (Funzione di proiezione $(x)_i$)**.** Definiamo la funzione primitiva ricorsiva $(x)_i$ in questo modo:

$$\text{se } x = [a_1, \cdots, a_n], \text{ allora } (x)_i = a_i.$$

Cioè,

$$(x)_i = \min_{t \leqslant x} \left(\neg (p_i^{t+1} | x) \right).$$

4.2.2 La funzione lunghezza $Lt(x)$

Definizione 4.5 (Funzione lunghezza $Lt(x)$)**.** La funzione primitiva ricorsiva "lunghezza" $Lt(x)$ è definita come

$$Lt(x) = \min_{t \leqslant x} \left[(x)_t \neq 0 \wedge (\forall y)_{\leqslant x} (y \leqslant t \vee (x)_y = 0) \right]$$

Ad esempio, si ha $Lt(36) = Lt(2^2 \cdot 3^2) = 2$, $Lt(25) = 3$, $Lt(0) = Lt(1) = 0$, $Lt(1024) = 1$.

Teorema 4.3 (Sequence number theorem)**.** *Riepilogando le proprietà delle funzioni finora descritte, valgono le seguenti*

a. $([a_1, \cdots, a_n])_i = \begin{cases} a_i & \text{se } 1 \leqslant i \leqslant n \\ 0 & \text{altrimenti} \end{cases}$

b. $[(x)_1, \cdots, (x)_n] = x$ *se* $n \geqslant Lt(x)$.

Capitolo 5

Un programma universale

One Program to rule them all, One Program to find them, One Program to bring them all and in the darkness bind them [...]

Alan Turing

5.1 Codificare programmi usando numeri

Il nostro obbiettivo è quello di associare ad ogni programma $\mathscr{P}$ nel linguaggio $\mathscr{S}$ un numero, che indicheremo con $\#(\mathscr{P})$, in modo tale che da esso sia possibile risalire al programma $\mathscr{P}$.

Definizione 5.1 (Codifica numerica di variabili e etichette)**.** Iniziamo stabilendo un ordinamento per le variabili

$$\underset{1}{Y}\ \underset{2}{X_1}\ \underset{3}{Z_1}\ \underset{4}{X_2}\ \underset{5}{Z_2}\ \underset{6}{X_3}\ \cdots$$

ed uno per le etichette

$$\underset{1}{A_1}\ \underset{2}{B_1}\ \underset{3}{C_1}\ \underset{4}{D_1}\ \underset{5}{E_1}\ \underset{6}{A_2}\ \underset{7}{B_2}\ \cdots$$

Quindi indicheremo con $\#(V)$ il numero associato alla posizione di V nella lista di variabili e con $\#(L)$ il numero associato alla posizione dell'etichetta nella nostra lista.

Ad esempio $\#(X_3) = 6$ e $\#(B_2) = 7$.

5.1.1 Codifica numerica di $\mathscr{S}$-Istruzioni

Definizione 5.2 (Codifica numerica di istruzioni)**.** Sia I una $\mathscr{S}$-Istruzione, allora indicheremo la codifica di I con

$$\#(I) = \langle a, \langle b, c\rangle\rangle$$

dove

1. Se I è etichettata con L, allora $a = \#(L)$; altrimenti $a = 0$;
2. Se la variabile V è menzionata in I, allora $c = \#(V) - 1$;
3. Se l'asserzione in I è un'istruzione pigra, un incremento o un decremento, allora b sarà uguale, rispettivamente, a 0, 1 oppure 2.
4. Se l'assezione in I è un salto condizionato IF $V \neq 0$ GOTO L^*, allora $b = \#(L^*) + 2$.

Come semplice esempio, si consideri l'istruzione $I =$[A] $Z \leftarrow Z - 1$. La codifica di I sarà

$$\#(I) = \langle a, \langle b, c\rangle\rangle = \langle 1, \langle 1, 2\rangle\rangle = \langle 1, 9\rangle = 37.$$

Decodifica di un'istruzione Supponiamo di conoscere il numero $\#(I) = \langle a, \langle b, c\rangle\rangle$. Come possiamo risalire all'istruzione I?
È sufficiente utilizzare le funzioni l e r già definite in 4.2 in questo modo:

Qual è l'etichetta?	$l(\#(I))$
Qual è la variabile menzionata?	$r(r((\#(I))) + 1$
Qual è il tipo d'istruzione/etichetta (+2) per il salto?	$l(r((\#(I)))$

5.1.2 Codifica numerica di un Programma

Definizione 5.3 (Codifica numerica di un S-Programma)**.** Sia $\mathscr{P}$ un S-Programma che consiste nella sequenza di istruzioni $I_1, \cdots, I_k$. Allora definiamo

$$\#(\mathscr{P}) = [\#(I_1), \cdots, \#(I_k)] - 1.$$

Nota. Sottraiamo 1 per mettere i programmi in corrispondenza biunivoca con $\mathbb{N}$.

Una pericolosa ambiguità La codifica dell'istruzione pigra priva di etichetta ($Y \leftarrow Y$) è

$$\#(I) = \langle 0, \langle 0, 0\rangle\rangle = 0$$

e ciò metterebbe in crisi la nostra codifica dei programmi, a causa dell'ambiguità delle codifiche di Gödel che abbiamo già notato in 4.2. Per ovviare a questo problema, imporremo che nessun programma legittimo possa terminare con l'istruzione $Y \leftarrow Y$, cosa che, chiaramente, non impone affatto restrizioni alla "potenza" del linguaggio. Dopo l'imposizione di questo vincolo, possiamo affermare che ad ogni numero naturale corrisponderà un programma e viceversa.

5.1.3 Esercizio 2, p. 67 dal *Davis* - Trovare $\mathscr{P}$ tale che $\#(\mathscr{P})$=575.

Per mettere in pratica quanto finora detto, svolgiamo un esercizio dal libro di testo come da titolo. Innanzitutto fattorizziamo $575 + 1 = 576 = 2^6 \cdot 3^2$, dunque il programma $\mathscr{P}$ è formato da due istruzioni tali che $\#(I_1) = 6$ e $\#(I_1) = 2$. Troviamo le due istruzioni:
$\langle a_1, \langle b_1, c_1\rangle\rangle = 6 \Leftrightarrow a_1 = 0,\ \langle b_1, c_1\rangle = 3 \Leftrightarrow a_1 = 0,\ b_1 = 2,\ c_1 = 0.$
$\langle a_2, \langle b_2, c_2\rangle\rangle = 2 \Leftrightarrow a_2 = 0,\ \langle b_2, c_2\rangle = 1 \Leftrightarrow a_2 = 0,\ b_2 = 1,\ c_2 = 0.$
Il programma che cerchiamo, dunque, è il seguente

```
1    Y ← Y − 1
2    Y ← Y + 1
```

ed effettua un semplice incremento di Y.

5.2 Esistono funzioni che non sono parzialmente calcolabili

Da quanto visto finora, abbiamo stabilito una corrispondenza biunivoca tra i programmi e l'insieme dei numeri naturali. I programmi possibili sono un'*infinità numerabile* e tale ordinamento induce una numerazione sulle funzioni di una variabile calcolate, come mostra la tabella seguente.

Numeri naturali	0	1	$\cdots$	i	$\cdots$	
	$\downarrow$	$\downarrow$		$\downarrow$		
Programma	$\mathscr{P}_0$	$\mathscr{P}_1$	$\cdots$	$\mathscr{P}_i$	$\cdots$	$\leftarrow$ non contiene ripetizioni
	$\downarrow$	$\downarrow$		$\downarrow$		
Funzione	$\psi_0^1(x)$	$\psi_1^1(x)$	$\cdots$	$\psi_i^1(x)$	$\cdots$	$\leftarrow$ contiene infinite ripetizioni

Teorema 5.1 (Esiste una funzione non parzialmente calcolabile)**.** *Esiste una funzione che non è parzialmente calcolabile, cioè che non è calcolabile da S-Programmi.*

Dimostrazione. Procediamo ancora una volta per diagonalizzazione. Consideriamo la funzione

$$g(x) = \begin{cases} \uparrow & \text{se } \psi_x(x) \downarrow \\ \downarrow & \text{se } \psi_x(x) \uparrow \end{cases}$$

g non può essere nessuna delle funzioni precedentemente enumerate, perchè differirà sempre sulla diagonale da ciascuna di esse! La seguente tabella chiarirà meglio il concetto:

	0	1	2	$\cdots$
$\psi_0(x)$	$\psi_0(0)$	$\psi_0(1)$	$\psi_0(2)$	$\cdots$
$\psi_1(x)$	$\psi_1(0)$	$\psi_1(1)$	$\psi_1(2)$	$\cdots$
$\psi_2(x)$	$\psi_2(0)$	$\psi_2(1)$	$\psi_2(2)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Dal momento che nella tabella abbiamo elencato tutte le funzioni calcolate dall'infinità numerabile di S-Programmi e che g non vi può appartenere, concludiamo che g non è parzialmente calcolabile. □

5.3 Il problema della fermata

Definizione 5.4 (Il predicato HALT)**.**

$HALT(x, y) \Leftrightarrow$ il programma numero y si ferma con input x.

Teorema 5.2 (Il predicato HALT non è calcolabile)**.** *Il predicato HALT precedentemente definito non è calcolabile.*

Dimostrazione. Supponiamo per assurdo che $HALT(x, y)$ sia calcolabile. Allora possiamo costruire il programma $\mathscr{P}$

```
1 [A]    IF HALT(X,X) GOTO A
```

Chiaramente si ha

$$\psi^1_{\mathscr{P}}(x) = \begin{cases} \uparrow & \text{se } \quad HALT(x, x) \\ 0 & \text{se } \neg HALT(x, x). \end{cases}$$

Supponiamo che il numero del programma precedentemente descritto sia $\#(\mathscr{P}) = y_0$. Allora usando la definizione del predicato $HALT$ si ha

$$HALT(x, y_0) \Leftrightarrow \neg HALT(x, x), \quad \forall x.$$

Dal momento che quanto scritto sopra dev'essere valido per ogni x, sarà valido anche per $x = y_0$. In tal caso però si avrebbe

$$HALT(y_0, y_0) \Leftrightarrow \neg HALT(y_0, y_0),$$

e la contraddizione che ne segue non può che significare che l'assunzione iniziale era errata, dunque provando il teorema. □

La tesi di Church-Turing

Non esiste un algoritmo che, dato un programma ed un suo input, possa determinare se il programma terminerà o meno con quel dato input.

Se la tesi di Church-Turing fosse falsa, allora detto algoritmo esisterebbe e, poichè abbiamo motivo di credere che ogni algoritmo che operi su numeri possa essere "implementato" nel linguaggio $\mathscr{S}$, esisterebbe un S-Programma che effettui la verifica. Ciò contraddirrebbe il fatto che $HALT$ non è un predicato calcolabile.

5.4 Universalità

Definizione 5.5 (Funzione universale Φ)**.** Definiamo, per ogni $n > 0$, la funzione

$$\Phi^{(n)}(x_1, \cdots, x_n, y) = \psi^n_{\mathscr{P}}(x_1, \cdots, x_n) \text{ dove } \#(\mathscr{P}) = y.$$

Teorema 5.3 (Teorema di universalità)**.** *Per ogni $n > 0$, la funzione $\Phi^{(n)}(x_1, \cdots, x_n, y)$ è parziamente calcolabile.*

Dimostrazione. Per dimostrare il teorema costruiamo, per ogni $n > 0$, un programma $\mathscr{U}_n$ che calcoli $\Phi^{(n)}$. Quindi si avrà, per ogni $n > 0$,

$$\psi^{(n+1)}_{\mathscr{U}_n}(x_1, \cdots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \cdots, x_n, x_{n+1})$$

Il programma universale si comporterà come un interprete. Dovrà decodificare il programma dato in ingresso, tenere traccia dello stato della computazione e dell'istruzione da eseguire, ed eseguire le istruzioni correnti aggiornando lo stato.
Codificheremo lo stato (sequenza di equazioni) usando la codifica di Gödel e l'ordine delle variabili stabilito in 5.1. Ad esempio, lo stato

$$Y = 1,\ X_1 = 2,\ X_2 = 1$$

è codificato dal numero

$$[1, 2, 0, 1] = 2^1 \cdot 3^2 \cdot 5^0 \cdot 7^1 = 126.$$

Notiamo fin d'ora che le variabili in ingresso si trovano nelle posizioni pari della nostra lista. Per il programma universale che stiamo per esibire ci discosteremo leggermente nei nomi delle variabili e delle etichette dalla sintassi del linguaggio $\mathscr{S}$ per motivi di chiarezza. In particolare, la variabile S conterrà lo stato del programma mentre K conterrà la posizione della prossima istruzione.

5.4.1 Il programma universale $\mathscr{U}_n$

Mostriamo quindi, procedendo per parti ed usando liberamente le funzioni primitive ricorsive finora incontrate, il programma $\mathscr{U}_n$ che calcola $Y = \Phi^{(n)}(x_1, \cdots, x_n, x_{n+1})$

```
1        Z ← X_{n+1} + 1
2        S ← ∏_{i=1}^{n} (p_{2i})^{x_i}
3        K ← 1
```

Dopo questa prima parte, Z conterrà $\#(\mathscr{P}) + 1$, ovvero $[\#(I_1), \cdots, \#(I_m)]$, S conterrà la codifica dello stato iniziale mentre K indicherà come prossima istruzione da eseguire la prima.

```
4  [C]  IF  K = Lt(Z) + 1 ∨ K = 0  GOTO  F
```

Se la computazione è terminata, salta ad F, dove sarà fornito in output il valore appropriato. Altrimenti prosegue "fetchando", decodificando e quindi eseguendo la prossima istruzione:

```
5        U ← r((Z)_k)
6        P ← p_{r(U)+1}
```

$(Z)_k = \#(I_k) = \langle a, \langle b, c\rangle\rangle$, quindi, dopo queste istruzioni, $U = \langle b, c\rangle$ e P sarà il numero primo il cui ordinamento nella lista è uguale alla posizione della variabile coinvolta nell'istruzione, ovvero sarà il primo il cui esponente nella fattorizzazione di S è il valore corrente della variabile. Proseguiamo:

```
7        IF  l(U) = 0  GOTO N
8        IF  l(U) = 1  GOTO A
9        IF  ¬(P|S)  GOTO N
10       IF  l(U) = 2  GOTO M
```

Se $l(U) = b = 0$, allora si tratta di un'istruzione pigra, e il programma non deve fare niente e quindi va a N (che potrebbe stare per *Next*). Se $b = 1$, invece, salta all'etichetta A (per *Add*). Se non si è effettuato nessuno dei salti precedenti, allora l'istruzione può essere soltanto un decremento (se $b = 2$), oppure un salto condizionato. In entrambi i casi, se $\neg(P|S)$, non si dovrà effettuare nessuna operazione (in quanto la variabile coinvolta avrà esponente 0 nella fattorizzazione di S, e quindi valore 0). Se non si effettua questo salto a *Null*, controlliamo se c'è da fare un decremento e, nel caso sia $b = 2$, si salta ad M (per *Minus*). Continuando:

```
11       K ← min_{i⩽Lt(Z)} [l((Z)_i) + 2 = l(U)]
12       GOTO C
```

Se giungiamo a queste righe di codice, allora $b > 2$ e $P|S$, quindi si dovrà (tentare di) effettuare un salto all'etichetta di numero $b - 2$. La minimalizzazione limitata dalla lunghezza del programma ritornerà in K (che contiene la posizione della prossima istruzione da eseguire) la posizione della prima riga etichettata con l'etichetta del salto. Se non trova alcuna istruzione con l'etichetta giusta, ritornerà 0. Quindi si tornerà in [C]. Continuando:

```
13 [M]   S ← ⌊S/P⌋
14       GOTO N
15 [A]   S ← S · P
16 [N]   K ← K + 1
17       GOTO C
```

Le istruzioni marcate da M e A, rispettivamente, decrementano e incrementano la variabile coinvolta nell'istruzione e sono seguite, in ogni caso, dall'istruzione che incrementa il contatore di programma e torna al loop principale in C. Manca soltanto l'ultima istruzione, quella con label [F], che copia il valore corretto in Y.

```
18 [F]   Y ← (S)_1
```

Con questo termina anche la nostra dimostrazione del teorema 5.3. □

```
1        Z ← X_{n+1} + 1
2        S ← ∏_{i=1}^{n} (p_{2i})^{x_i}
3        K ← 1
4  [C]   IF  K = Lt(Z) + 1 ∨ K = 0  GOTO F
5        U ← r((Z)_k)
6        P ← p_{r(U)+1}
7        IF  l(U) = 0  GOTO N
8        IF  l(U) = 1  GOTO A
9        IF  ¬(P|S)  GOTO N
10       IF  l(U) = 2  GOTO M
11       K ← min_{i⩽Lt(Z)} [l((Z)_i) + 2 = l(U)]
12       GOTO C
13 [M]   S ← ⌊S/P⌋
14       GOTO N
15 [A]   S ← S · P
16 [N]   K ← K + 1
17       GOTO C
18 [F]   Y ← (S)_1
```

Codice 5.1: Il programma universale $\mathscr{U}_n$

5.4.2 Il predicato STP

Definizione 5.6 (Il predicato STP)**.**
$STP^{(n)}(x_1, \cdots, x_n, y, t) \Leftrightarrow$ Il programma di numero y termina entro alpiù t passi su ingresso $(x_1, \cdots, x_n)$.
STP è calcolabile (basta modificare leggermente $\mathscr{U}_n$ aggiungendo un contatore di passi), ma dimostreremo successivamente (in 5.4) che STP è anche primitivo ricorsivo.

Teorema 5.4 (Teorema del conta-passi)**.**
Per ogni $n > 0$, il predicato $STP^{(n)}(x_1, \cdots, x_n, y, t)$ è primitivo ricorsivo.

Dimostrazione.
Incominciamo descrivendo alcune funzioni per estrarre i componenti dell'$i - esima$ istruzione dal programma di numero y:

$$LABEL(i, y) = l((y+1)_i)$$
$$VAR(i, y) = r(r((y+1)_i)) + 1$$
$$INSTR(i, y) = l(r((y+1)_i))$$
$$LABEL'(i, y) = l(r((y+1)_i)) \dot{-} 2.$$

Codificheremo lo stato del programma in z usando la stessa tecnica usata in 5.3. L'istantanea (i, σ) sarà codificata da $\langle i, z\rangle$. Quindi definiamo dei predicati che indichino, dato un programma y e un'istantanea $x = \langle i, z\rangle$, quale sarà la prossima azione da eseguire:

$$\begin{aligned}
SKIP(x, y) &= [INSTR(l(x), y) = 0 \wedge l(x) \leqslant Lt(y+1)] \vee \left[INSTR(l(x), y) \geqslant 2 \wedge \neg(p_{\mathrm{VAR}(l(x),y)} | r(x))\right] \\
INCR(x, y) &= INSTR(l(x), y) = 1 \\
DECR(x, y) &= INSTR(l(x), y) = 2 \wedge (p_{\mathrm{VAR}(l(x),y)} | r(x)) \\
BRANCH(x, y) &= INSTR(l(x), y) > 2 \wedge p_{\mathrm{VAR}(l(x),y)} | r(x) \wedge (\exists i)_{\leqslant Lt(y+1)} LABEL(i, y) = LABEL'(l(x), y)
\end{aligned}$$

Ora possiamo definire $SUCC(x, y)$ che, dato un programma y e un'istantanea x ritorna l'istantanea successore di x.

$$SUCC(x, y) = \begin{cases}
\langle l(x)+1, r(x)\rangle & \text{se } SKIP(x, y) \\
\langle l(x)+1, r(x) \cdot p_{\mathrm{VAR}(l(x),y)}\rangle & \text{se } INCR(x, y) \\
\langle l(x)+1, \lfloor r(x)/p_{\mathrm{VAR}(l(x),y)}\rfloor\rangle & \text{se } DECR(x, y) \\
\langle min_{i \leqslant Lt(y+1)}[LABEL(i, y) = LABEL'(l(x), y)], r(x)\rangle & \text{se } BRANCH(x, y) \\
\langle Lt(y+1)+1, r(x)\rangle & \text{altrimenti}
\end{cases}$$

Definiamo anche

$$INIT^{(n)}(x_1, \cdots, x_n) = \langle 1, \prod_{i=1}^{n} p_{2n}^{x_i}\rangle,$$

che dati i valori delle variabili iniziali ritorna lo stato iniziale e

$$TERM(x, y) \Leftrightarrow l(x) > Lt(y+1),$$

che verifica se un'istantanea x è un'istantanea terminale del programma y. Usando le funzioni primitive ricorsive finora definite, si può definire una funzione primitiva ricorsiva che calcoli via via le istantanee di un programma:

$$SNAP^{(n)}(x_1, \cdots, x_n, y, 0) = INIT^{(n)}(x_1, \cdots, x_n)$$
$$SNAP^{(n)}(x_1, \cdots, x_n, y, i+1) = SUCC(SNAP(^{(n)}(x_1, \cdots, x_n, y, i), y)$$

Quindi

$$STP(x_1, \cdots, x_n, y, t) \Leftrightarrow TERM(SNAP^{(n)}(x_1, \cdots, x_n, y, t), y)$$

e STP, poichè composizione di funzioni e predicati primitivi ricorsivi, è primitivo ricorsivo. □

5.4.3 Caratterizzazioni

Teorema 5.5 (Teorema di forma normale)**.** *Sia $f(x_1, \cdots, x_n)$ una funzione parzialmente calcolabile. Allora esiste un predicato primitivo ricorsivo $R(x_1, \cdots, x_n, y)$ tale che*

$$f(x_1, \cdots, x_n) = l\left(\underset{z}{min}\ R(x_1, \cdots, x_n, z)\right).$$

Dimostrazione. Sia $f(x_1, \cdots, x_n)$ parzialmente calcolabile e sia y_0 il numero del programma che la calcola. Sia R il predicato

$$STP^{(n)}(x_1, \cdots, x_n, y_0, r(z)) \wedge (r(SNAP^{(n)}(x_1, \cdots, x_n, y_0, r(z))))_1 = l(z).$$

Consideriamo $z = \langle l(z), r(z)\rangle$ dove $r(z)$ rappresenta il numero di passi e $l(z)$ il valore funzionale di f. La parte a sinistra del predicato R è soddisfatta dall'istante $r(z)$ in cui y_0 termina e da tutti gli istanti successivi. La parte destra richiede che nella variabile d'uscita Y vi sia il valore atteso di uscita $l(z)$. Ognuno degli z che soddisfa R soddisfa contemporaneamente entrambe queste condizioni. Se la parte a destra non viene mai verificata, sarà il caso in cui $STP^{(n)}(x_1, \cdots, x_n, y_0, t)$ è falso per ogni t, e quindi $f(x_1, \cdots, x_n) \uparrow$. Essendo $l(z)$ fissato, la minimalizzazione agisce sugli $r(z)$ selezionando il primo istante in cui il programma è terminato. □

Definizione 5.7 (Funzione parziale ricorsiva)**.** Una funzione f è parziale ricorsiva se e solo se può essere ottenuta dalle funzioni iniziali applicando un numero finito di volte gli schemi di composizione, ricorsione primitiva e minimalizzazione.

Teorema 5.6 (Teorema di caratterizzazione)**.** *Una funzione f è parzialmente calcolabile se e solo se si può ottenere dalle funzioni iniziali applicando un numero finito di volte composizione, ricorsione primitiva, minimalizzazione non limitata.*

Dimostrazione non fatta in classe. Ogni funzione che può essere ottenuta come descritto sopra è parzialmente calcolabile, come conseguenza dei teoremi 5.3, 3.1, 3.2, 3.3, 3.10. Per quanto riguarda l'altro verso dell'implicazione, il teorema di forma normale 5.5 ci garantisce che ogni funzione parzialmente calcolabile si può scrivere come

$$l(\underset{y}{min}\ R(x_1, \cdots, x_n))$$

dove R è un predicato primitivo ricorsivo, e quindi per definizione ottenuto dalle funzioni iniziali usando un numero finito di volte composizione e ricorsione. Ad R quindi applichiamo una minimalizzazione limitata e una composizione con la funzione primitiva ricorsiva l. □

Definizione 5.8 (Minimalizzazione propria)**.** Quando $min_y\ R(x_1, \cdots, x_n, y)$ è una funzione totale (cioè quando per ogni $x_1, \cdots, x_n$ c'è almeno una y tale che $R(x_1, \cdots, x_n, y)$ è vera), si dice che si aplica una minimalizzazione propria ad R. Quindi, se

$$l(\underset{y}{min}\ R(x_1, \cdots, x_n))$$

è totale, allora anche $min_y\ R(x_1, \cdots, x_n, y)$ dev'essere totale. Dunque abbiamo il seguente

Teorema 5.7 (Caratterizzazione delle funzioni calcolabili)**.** *Una funzione è calcolabile se e solo se può essere ottenuta dalle funzioni iniziali usando un numero finito di volte composizione, ricorsione e minimalizzazione propria (cioè è ricorsiva).*

Definizione 5.9 (Funzione ricorsiva)**.** Una funzione f è ricorsiva se e solo se può essere ottenuta dalle funzioni iniziali applicando un numero finito di volte gli schemi di composizione, ricorsione primitiva e minimalizzazione propria.

Capitolo 6

Insiemi ricorsivamente enumerabili

Definizione 6.1 (Insieme primitivo ricorsivo)**.** Un insieme S è primitivo ricorsivo se la sua funzione caratteristica f_s è primitiva ricorsiva.

Definizione 6.2 (Insieme calcolabile o ricorsivo)**.** Un insieme S è calcolabile o ricorsivo se la sua funzione caratteristica f_s è calcolabile.

Definizione 6.3 (Insieme ricorsivamente enumerabile)**.** Un insieme S è ricorsivamente enumerabile se esiste una funzione f unaria e parzialmente calcolabile tale che $S = \{x | f(x) \downarrow\}$, ovvero se esiste una funzione parzialmente calcolabile tale che S sia il suo insieme di definizione.

Teorema 6.1 (S ricorsivo $\Rightarrow$ S ricorsivamente enumerabile)**.** *Dimostrazione.* Sia S ricorsivo. Allora la sua funzione caratteristica f_s è calcolabile e, quindi, esisterà un S-Programma che la calcola. Per mostrare che S è ricorsivamente enumerabile dobbiamo trovare una funzione unaria parzialmente calcolabile che converge solo sugli elementi dell'insieme. Dal momento che f_s è calcolabile, il seguente S-Programma $\mathscr{P}$ è legittimo:

```
1 [A]    IF  f_s(x_1) = 0  GOTO  A
```

Il programma $\mathscr{P}$ calcola la funzione

$$g(x) = \begin{cases} 0 & \text{se } x \in S \\ \uparrow & \text{altrimenti} \end{cases},$$

quindi $S = \{x | g(x) \downarrow\}$, S ricorsivamente enumerabile. □

Teorema 6.2 (B ricorsivo $\Leftrightarrow$ B e $\overline{B}$ r.e.)**.** *Un insieme B è ricorsivo se e solo se B e $\overline{B}$ sono ricorsivamente enumerabili.*

Dimostrazione. Mostriamo la validità di entrambi i versi:
Solo se $\Longrightarrow$ Sia B un insieme ricorsivo. Allora al sua funzione caratteristica è un predicato calcolabile e, per il teorema 3.8, anche $\overline{B}$ sarà ricorsivo. Quindi, per il teorema 6.1, sono entrambi ricorsivamente enumerabili.
Se $\Longleftarrow$ Siano B e $\overline{B}$ ricorsivamente enumerabili. Allora esisteranno le funzioni parzialmente calcolabili

$$f \ : \ B = \{x | f(x) \downarrow\} \Rightarrow \exists \mathscr{P}_1 \text{ che calcola } f \ : \ \#(\mathscr{P}_1) = p$$
$$g \ : \ \overline{B} = \{x | g(x) \downarrow\} \Rightarrow \exists \mathscr{P}_2 \text{ che calcola } g \ : \ \#(\mathscr{P}_2) = q$$

Allora il seguente programma calcola la funzione caratteristica di B $h_B(x) = \begin{cases} 1 & \text{se } x \in B \\ 0 & \text{se } x \notin B \text{ cioè } x \in \overline{B} \end{cases}$

```
1  [A]   IF STP(X, p, Z) GOTO B
2        IF STP(X, q, Z) GOTO E
3        Z ← Z + 1
4        GOTO A
5  [B]   Y ← Y + 1
```

□

6.1 Proprietà di chiusura

Teorema 6.3 (Chiusura degli insiemi r.e. rispetto a $\cup$, $\cap$)**.** *Siano gli insiemi B e C ricorsivamente enumerabili. Allora anche $B \cup C$ e $B \cap C$ sono ricorsivamente enumerabili.*

Dimostrazione. Dall'ipotesi deduciamo che esistono le seguenti funzioni unarie e parzialmente calcolabili:

$$f \ : \ B = \{x | f(x) \downarrow\} \Rightarrow \exists \mathscr{P}_1 \text{ che calcola } f \ : \ \#(\mathscr{P}_1) = p$$

$$g \ : \ C = \{x | g(x) \downarrow\} \Rightarrow \exists \mathscr{P}_2 \text{ che calcola } g \ : \ \#(\mathscr{P}_2) = q$$

Consideriamo il seguente programma

```
1        Z ← f(x)
2        Z ← g(x)
```

che calcola $h(x) = \begin{cases} 0 & \text{se } x \in B \cap C \\ \uparrow & \text{altrimenti} \end{cases}$.
Da ciò segue che $B \cap C$ è ricorsivamente enumerabile ($B \cap C = \{x | h(x) \downarrow\}$).

Esibiamo invece un programma che calcola una funzione unaria che converge soltanto per gli elementi di $B \cup C$:

```
1  [A]   IF STP(x, p, Z) GOTO E
2        IF STP(x, q, Z) GOTO E
3        Z ← Z + 1
4        GOTO A
```

Il programma appena mostrato calcola $j(x) = \begin{cases} 0 & \text{se } x \in B \cup C \\ \uparrow & \text{altrimenti} \end{cases}$.
$B \cup C = \{x | j(x) \downarrow\}$, mostrando che $B \cup C$ è ricorsivamente enumerabile e provando il teorema. □

6.2 Il teorema di enumerazione

Definizione 6.4 (W_n)**.** Definiamo, per ogni $n \geqslant 0$,

$$W_n = \{x | \Phi(x, n) \downarrow\}.$$

W_n è ricorsivamente enumerabile in quanto Φ è una funzione unaria (n è costante) e parzialmente calcolabile dal programma $\mathscr{U}_n$, come mostrato nel teorema 5.3.

Teorema 6.4 (Teorema di enumerazione)**.** *Un insieme B è ricorsivamente enumerabile se e solo se esiste una n per cui $B = W_n$.*

Dimostrazione. Dal momento che i programmi sono numerabili, possiamo indurre un ordinamento anche sugli insiemi ricorsivamente enumerabili. Ovvero

Numeri naturali	0	1	$\cdots$	i	$\cdots$	
	$\downarrow$	$\downarrow$		$\downarrow$		
Programma	$\mathscr{P}_0$	$\mathscr{P}_1$	$\cdots$	$\mathscr{P}_i$	$\cdots$	$\leftarrow$ non contiene ripetizioni
	$\downarrow$	$\downarrow$		$\downarrow$		
Funzione	$\psi_0^1(x)$	$\psi_1^1(x)$	$\cdots$	$\psi_i^1(x)$	$\cdots$	$\leftarrow$ contiene infinite ripetizioni
	$\downarrow$	$\downarrow$		$\downarrow$		
Insiemi r.e.	W_0	W_1	$\cdots$	W_i	$\cdots$	$\leftarrow$ contiene infinite ripetizioni

Tale numerazione è esaustiva, ma presenta infinite ripetizioni. Infatti, dato un W_i, ci sono infiniti W_j che corrispondono allo stesso insieme (perchè ci sono infiniti programmi che calcolano la stessa funzione). □

Corollario 6.4.1 ($\mathbb{N}$ è ricorsivamente enumerabile)**.** *Consideriamo, ricordando che 0 è associato al programma vuoto, l'insieme*

$$W_0 = \{x | \Phi(x, 0) \downarrow\} = \mathbb{N}$$

Quindi l'insieme dei numeri naturali è ricorsivamente enumerabile.

Definizione 6.5 (Insieme diagonale K)**.**
Sia

$$K = \{n \in \mathbb{N} | \Phi(n, n) \downarrow\}.$$

Si ha

$$n \in W_n \Leftrightarrow \Phi(n, n) \downarrow \Leftrightarrow HALT(n, n).$$

Teorema 6.5 (K è ricorsivamente enumerabile)**.**

Dimostrazione. Dal momento che Φ è una funzione parzialmente calcolabile (teorema 5.3), K è ricorsivamente enumerabile. □

6.2.1 Esistono insiemi non ricorsivamente enumerabili

Teorema 6.6 ($\overline{K}$ non è ricorsivamente enumerabile)**.**

Dimostrazione. Supponiamo per assurdo che

$$\overline{K} = \{x | \Phi(x, x) \uparrow\}$$

sia ricorsivamente enumerabile. Allora per il teorema 6.4, si avrebbe un i tale che

$$\overline{K} = W_i.$$

Si avrebbe quindi che

$$x \in \overline{K} \Leftrightarrow \Phi(x, i) \downarrow .$$

In particolare, per $x = i$ si ha

$$i \in \overline{K} \Leftrightarrow \Phi(i, i) \downarrow \Leftrightarrow i \in K.$$

Ciò è chiaramente assurdo dal momento che un insieme ed il suo complemento sono disgiunti. □

Corollario 6.6.1 (K non è ricorsivo)**.**
Dai teoremi 6.6 e 6.2 segue banalmente che K non può essere ricorsivo, dal momento che $\overline{K}$ non è ricorsivamente enumerabile. K ammette soltanto un processo di semidecisione (miglior risultato possibile per K).

Teorema 6.7. *Sia B un insieme ricorsivamente enumerabile. Allora esiste un predicato primitivo ricorsivo $R(x,t)$ tale che $B = \{x \in \mathbb{N} | (\exists t)\ R(x,t)\}$.*

Dimostrazione non fatta in classe. Sia $B = W_n$ per qualche n. Allora $B = \{x \in \mathbb{N} | (\exists t)\ STP^{(1)}(x,n,t)\}$ ed STP è primitivo ricorsivo per il teorema 5.4. □

6.3 Altri teoremi

Teorema 6.8. *Sia S un insieme non vuoto ricorsivamente enumerabile. Allora esiste una funzione primitiva ricorsiva $f(u)$ tale che $S = \{f(n) | n \in \mathbb{N}\}$. S si dice **range** di f.*

Teorema 6.9. *Sia f una funzione parzialmente calcolabile e sia $S = \{f(x) | f(x)\downarrow\}$ (S è il range di f). Allora S è ricorsivamente enumerabile.*

Teorema 6.10 (Teorema di equivalenza)**.** *Sia $S \neq \varnothing$. Allora le seguenti affermazioni sono equivalenti:*

(i) S è ricorsivamente enumerabile;

(ii) S è il range di una funzione primitiva ricorsiva;

(iii) S è il range di una funzione ricorsiva;

(iv) S è il range di una funzione parziale ricorsiva;

Parte II

Automi

Capitolo 7

La macchina di Turing

7.1 La tesi di Church-Turing

Una nuova formulazione della tesi di Church-Turing già incontrata in 5.3:

Ogni funzione parzialmente calcolabile mediante un algoritmo è parzialmente calcolabile mediante una macchina di Turing.

Teorema 7.1 (Teoremi di equivalenza)**.** *Una funzione f è parzialmente calcolabile mediante S-programmi sse f è parziale ricorsiva sse è parzialmente calcolabile da una macchina di Turing.*

7.2 La macchina di Turing

Una macchina di Turing (abbreviato MdT) è una macchina astratta che opera su un nastro di memoria infinito. Dispone di una testina mobile che è in grado di leggere/scrivere/cancellare da tale nastro un simbolo per volta e di spostarsi di una cella a destra o a sinistra per leggere/scrivere/cancellare nuovi simboli. Più precisamente diamo la seguente definizione:

Definizione 7.1 (Macchina di Turing deterministica)**.** Una MdT deterministica a un nastro e istruzioni a cinque campi ha questa forma

$$T = \langle Q, q_1, F, A, \beta, \delta \rangle$$

dove

- $Q = \{q_1, \cdots, q_k\}$ è un insieme finito detto insieme degli stati della macchina;
- $q_1 \in Q$ è detto stato iniziale della macchina T;
- $F \subseteq Q$ è detto insieme degli stati finali della macchina T;
- $A = \{s_1, \cdots, s_m\}$ è un insieme finito di simboli detto alfabeto della macchina T;
- $\beta \in A$ è un simbolo di A detto segno di casella vuota di T;
- $\delta : S \times A \to S \times A \times \{L, F, R\}$ è detta funzione di transizione di T. δ è iniettiva e ciò ci garantisce un sistema deterministico;

Se consideriamo $\delta(q_i, s_j) \to \langle q_e, s_m, \S \rangle$, la corrispondente quintupla

$$\langle q_i, s_j, q_e, s_m, \S \rangle$$

può essere considerata come un'istruzione interpretata come segue:

$$\langle \overbrace{q_i, s_j}^{condizione} : \overbrace{q_e, s_m, \S}^{azione} \rangle$$

dove § rappresenta lo spostamento eventualmente intrapreso (L sinistra, R destra, F fermo).

Capitolo 8

Automi a stati finiti

8.1 Automi finiti deterministici

Un automa finito deterministico (DFA da *Deterministic Finite Automata*) ha le seguenti caratteristiche:

- La dimensione della memoria ("nastro") non cambia in corso d'opera ma è fissata dalla dimensione n dell'input.
- Non effettua scritture
- Ad ogni istante deve spostarsi di una casella a destra (e nel farlo può cambiare stato).
- L'automa accetta o respinge la stringa in input.

Più formalmente, definiamo un automa finito in questo modo:

Definizione 8.1 (Automa finito deterministico)**.** Un automa finito ha la seguente forma:

$$\mathscr{M} = \langle A, Q, F, q_1, \delta \rangle$$

dove

- $A = \{s_1, \cdots, s_k\}$ è un insieme finito di simboli detto alfabeto;
- $Q = \{q_1, \cdots, q_n\}$ è un insieme finito di stati dell'automa;
- $F \subseteq Q$ è detto insieme degli stati di accettazione dell'automa;
- $q_1 \in Q$ è uno stato di $\mathscr{M}$ detto stato iniziale;
- $\delta : \ Q \times A \to Q$ è detta funzione di transizione di $\mathscr{M}$.

Definizione 8.2 (A* sull'alfabeto A)**.** Sia A un alfabeto finito,

$$A^* = \{u \mid u = s_1, \cdots, s_n \text{ con } s_i \in A \text{ per ogni } 1 \leqslant i \leqslant n \text{ ed } n \in \mathbb{N}\}$$

Definizione 8.3 (Lunghezza di una stringa)**.** Se $u = s_1, \cdots, s_n$ è una stringa sull'alfabeto A, allora la lunghezza di u indicata con $|u|$ è n. La parola vuota ha lunghezza zero e viene indicata con il simbolo 0 oppure ε.

Definizione 8.4 (Funzione δ^*)**.** La funzione

$$\delta^* : Q \times A^* \to Q$$

dato uno stato iniziale ed una stringa ritorna lo stato in cui perviene l'automa dopo aver esaminato la stringa u partendo dallo stato q. Si può definire ricorsivamente in questo modo:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, ws_j) = \delta(\delta^*(q, w), s_j)$$

Definizione 8.5 (Linguaggio riconosciuto da $\mathscr{M}$)**.** Il linguaggio riconosciuto dall'automa $\mathscr{M}$ è

$$L(\mathscr{M}) = \{u \in A^* \mid \delta^*(q_1, u) \in F\}$$

Definizione 8.6 (Linguaggio regolare)**.** Il linguaggio $\mathcal{L} \subseteq A^*$ è un linguaggio regolare se e solo se esiste $\mathscr{M}$ automa finito deterministico (DFA) tale che $\mathcal{L} = L(\mathscr{M}) = \{u \in A^* \mid \delta^*(q_1, u) \in F\}$.

8.2 Alcuni esercizi esemplificativi

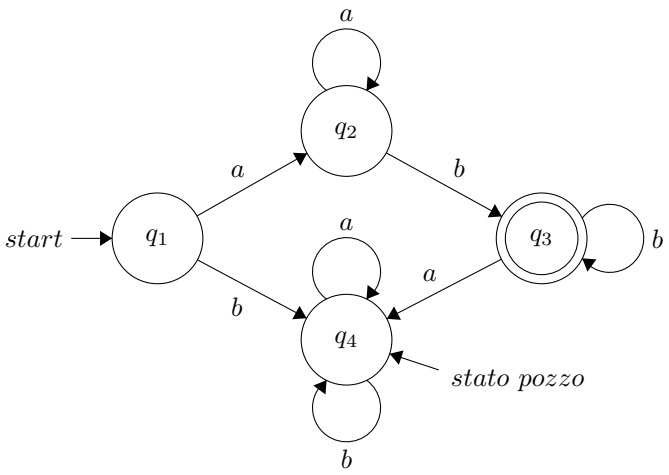
8.2.1 Un problema di analisi

Trovare il linguaggio riconosciuto dal seguente automa $\mathscr{M}_{\text{DFA}} = \langle A, Q, F, q_1, \delta \rangle$ dove

- $A = \{a, b\}$;
- $Q = \{q_1, q_2, q_3, q_4\}$;
- $F = \{q_3\}$;
- δ è definita dalla seguente tabella:

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

Per comodità rappresentiamo l'automa con un diagramma a bolle (in futuro saranno rappresentati esclusivamente in questo modo). Per convenzione indicheremo gli stati di accettazione con un doppio cerchio. Le frecce orientate che collegano i vari stati rappresentano una transizione da uno stato all'altro quando il carattere letto è quello che etichetta la freccia. Il diagramma sopra descritto viene rappresentato come



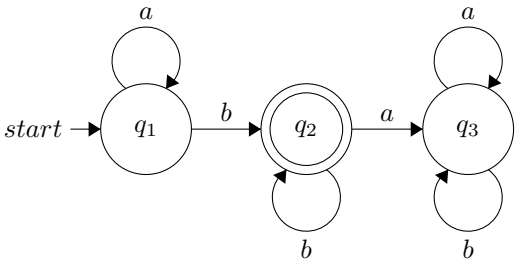
Dunque si ha che $L(\mathscr{M}) = \{a^{[n]}b^{[m]}, \text{ con } n, m > 0\}$, cioè $\mathscr{M}$ accetta tutte le stringhe che sono composte da un numero maggiore di 0 di a seguito da almeno una b. Tutte le altre stringhe saranno rifiutate.

8.2.2 Un problema di sintesi

Si descriva un automa $\mathscr{M}_{\text{DFA}}$ che riconosca il seguente linguaggio:

$$L = \{a^{[n]}b^{[m]} \mid n \geqslant 0, m > 0\}$$

Il seguente automa rapprentato con diagramma a bolle riconosce L:



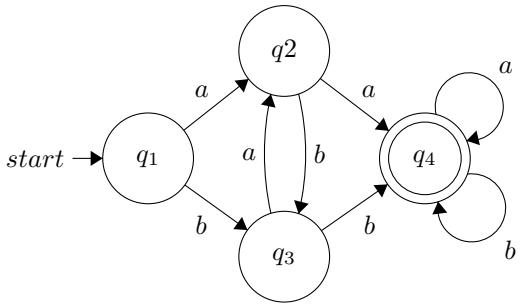
8.3 Automi finiti non deterministici

Definizione 8.7 (Automa finito non deterministico)**.** Definiamo un automa finito non deterministico (NDFA da *NonDeterministic Finite Automata*) semplicemente modificando la definizione di automa finito deterministico data in 8.1 in modo tale che sia la funzione di transizione

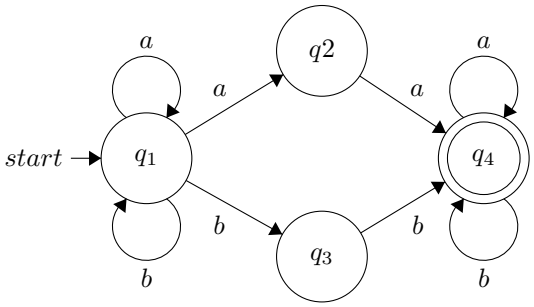
$$\delta \ : \ Q \times A \to P(Q),$$

dove con $P(Q)$ indichiamo l'insieme delle parti di Q, rendendo quindi possibili transizioni verso insiemi di stati (e anche verso nessuno stato, essendo $\varnothing \in P(Q)$).

Si consideri ad esempio il linguaggio $\mathcal{L} = \{u \in A^* \mid u = vaaw \vee u = vbbw, \ v, w \in A^*\}$. $\mathcal{L}$ corrisponde all'insieme delle parole con almeno due a o due b consecutive. Iniziamo dando un automa DFA che riconosca $\mathcal{L}$:



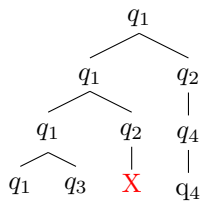
Quindi proviamo a semplificare l'automa DFA appena descritto realizzando un NDFA che riconosca lo stesso linguaggio:



Realizziamo la tabella di transizione dell'automa NDFA per chiarirne il comportamento:

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

Come si intuisce, in automi non deterministici la computazione non è lineare bensì parallela. L'albero di computazione qui rappresentato per l'input `aab` rende molto bene questo concetto:



La X indica che si compie una transizione verso l'insieme vuoto, terminando quel ramo di computazione. L'automa NDFA accetta la stringa se e solo se, terminata la computazione, l'insieme degli stati in cui si perviene non è disgiunto da F.

Definizione 8.8 (δ^* per automi non deterministici)**.** Diamo una definizione ricorsiva di δ^* per automi NDFA

$$\delta^*(q, 0) = \{q\}$$
$$\delta^*(q, vs_j) = \bigcup_{q_i \in \delta^*(q,v)} \delta(q_i, s_j)$$

Teorema 8.1 ($\mathcal{L}$ regolare $\Rightarrow$ $\mathcal{L}$ riconoscibile da NDFA)**.** *Se $\mathcal{L}$ è un linguaggio regolare, allora $\mathcal{L}$ è riconoscibile da un automa finito non deterministico.*

Dimostrazione. Se $\mathcal{L}$ è regolare, esisterà $\mathscr{M}_{\text{DFA}}$ che lo riconosce. Sia $\mathscr{M}'_{NDFA}$ ottenuto da $\mathscr{M}$ modificando la funzione δ nel seguente modo:

$$\delta'(q, s_i) = \{\delta(q, s_i)\}$$

$\mathscr{M}'_{NDFA}$ riconosce $\mathcal{L}$ provando il teorema. □

Teorema 8.2 ($\mathcal{L}$ riconosciuto da NDFA $\Rightarrow$ $\mathcal{L}$ regolare)**.** *La tesi equivale a: se un linguaggio è riconoscibile da un NDFA, allora esiste un DFA che lo riconosce.*

Dimostrazione. Sia il linguaggio $\mathcal{L}$ riconosciuto da $\mathscr{M}_{NDFA=\{A,Q,F,q_1,\delta\}}$. Specifichiamo un $\widetilde{\mathscr{M}}_{\text{DFA}} = \{\widetilde{A}, \widetilde{Q}, \widetilde{F}, \widetilde{Q1}, \widetilde{\delta}\}$ tale che $L(\mathscr{M}_{NDFA}) = L(\widetilde{\mathscr{M}}_{\text{DFA}})$.
Gli alfabeti dei due automi saranno uguali, quindi $\widetilde{A} = A$. Per ogni elemento di $P(Q)$, abbiamo uno stato in $\widetilde{Q} = \{Q_1, \cdots, Q_{2^n}\}$, con in particolare $Q_1 = \{q_1\}$. Gli stati di accettazione saranno $\widetilde{F} = \{Q_i \mid Q_i \cap F \neq \varnothing\}$. Lo stato iniziale sarà il già definito Q_1 mentre $\widetilde{\delta}$ è definita in questo modo:

$$\widetilde{\delta}(Q_i, s) = \bigcup_{q \in Q_i} (\delta(q, s))$$

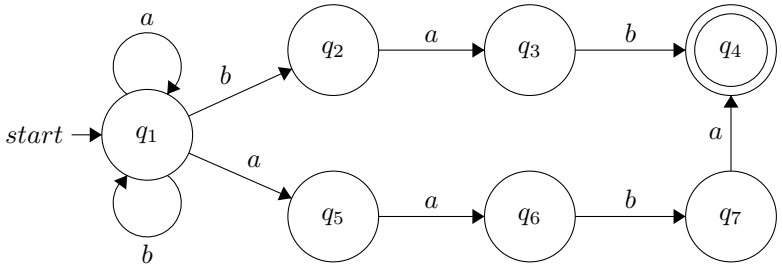
Con questo abbiamo terminato la costruzione del DFA. La dimostrazione di correttezza della specifica non è stata oggetto del corso. □

Teorema 8.3 (Un linguaggio è riconosciuto da un NDFA se e solo se è riconosciuto da un DFA)**.** *Un linguaggio è accettato da un NDFA se e solo se è regolare. Equivalentemente, un linguaggio è accettato da un NDFA se e solo se è accettato da un DFA.*

Dimostrazione. La tesi segue immediatamente dai teoremi 8.1 e 8.2. □

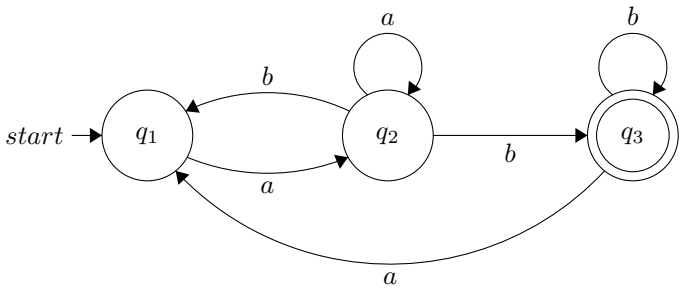
8.3.1 Esercizio di sintesi

Costruire un NDFA che riconosca $\mathcal{L} = \{uw \in A^* \mid w = bab \vee w = aaba\}$.



8.3.2 Altro esercizio di sintesi

Costruire un NDFA che riconosca $\mathcal{L} = \{a^{m_1}b^{n_1} \cdots a^{m_r}b^{n_r} \mid m_1, n_1, \cdots, m_r, n_r > 0\}$



8.4 Proprietà di chiusura

8.4.1 Automi DFA non-restarting

Definizione 8.9 (Automa finito non-restarting)**.** Un DFA è non-restarting se non vi è alcuna coppia (q, s) tale che $\delta(q, s) = q_1$.

Teorema 8.4 (Per ogni $\mathscr{M}_{\text{DFA}}$ esiste $\widetilde{\mathscr{M}}_{\text{DFA}}$ non-restarting tale che $L(\mathscr{M}) = L(\widetilde{\mathscr{M}})$)**.** *In altre parole, ogni linguaggio regolare è riconosciuto da un DFA non-restarting.*

Dimostrazione. Sia $\mathscr{M} = \langle A, Q, F, q_1, \delta\rangle$ un automa finito deterministico restarting, con $Q = \{q_1 \cdots, q_n\}$. Dimostriamo il teorema specificando l'automa finito deterministico $\widetilde{\mathscr{M}} = \langle \widetilde{A}, \widetilde{Q}, \widetilde{F}, \widetilde{q_1}, \widetilde{\delta}\rangle$ tale che $L(\mathscr{M}) = L(\widetilde{\mathscr{M}})$.

- $\widetilde{A} = A$
- $\widetilde{Q} = Q \cup \{q_{n+1}\} = \{q_1, \cdots, q_n, q_{n+1}\}$
- $\widetilde{F} = \begin{cases} F & \text{se } q_1 \notin F \\ F \cup \{q_{n+1}\} & \text{se } q_1 \in F \end{cases}$
- $\widetilde{q_1} = q_1$
- $\widetilde{\delta}(q, s) = \begin{cases} \delta(q, s) & \text{se } q \in Q \wedge \delta(q, s) \neq q_1 \\ q_{n+1} & \text{se } q \in Q \wedge \delta(q, s) = q_1 \end{cases}$
  $\widetilde{\delta}(q_{n+1}, s) = \widetilde{\delta}(q_1, s)$

La dimostrazione costruttiva esibita mostra anche l'algoritmo per ottenere un DFA non-restarting da un DFA restarting. □

8.4.2 Unione

Teorema 8.5 (La classe dei linguaggi regolari è chiusa rispetto all'unione)**.** *Se $\mathcal{L}$ e $\widetilde{\mathcal{L}}$ sono linguaggi regolari, allora $\mathcal{L} \cup \widetilde{\mathcal{L}}$ è un linguaggio regolare.*

Dimostrazione. Per il teorema 8.4 possiamo assumere senza perdità di generalità che sia

$$\mathcal{L} = L(\mathscr{M}) \text{ con } \mathscr{M} = \langle A, Q, F, q_1, \delta\rangle \text{ DFA non-restarting}$$
$$\widetilde{\mathcal{L}} = L(\widetilde{\mathscr{M}}) \text{ con } \widetilde{\mathscr{M}} = \langle A, \widetilde{Q}, \widetilde{F}, \widetilde{q_1}, \widetilde{\delta}\rangle \text{ DFA non-restarting.}$$

Assumendo che $Q \cap \widetilde{Q} = \varnothing$, specifichiamo un NDFA $\check{\mathscr{M}} = \langle A, \check{Q}, \check{F}, \check{q_1}, \check{\delta}\rangle$ tale che $L(\check{\mathscr{M}}) = \mathcal{L} \cup \widetilde{\mathcal{L}}$.

- $\check{Q} = Q \cup \widetilde{Q} \cup \{\check{q_1}\} - \{q_1, \widetilde{q_1}\}$
- $\check{F} = \begin{cases} F \cup \widetilde{F} \cup \{\check{q_1}\} & \text{se } q_1 \in F \text{ oppure } \widetilde{q_1} \in \widetilde{F} \\ F \cup \widetilde{F} & \text{altrimenti} \end{cases}$
- $\check{\delta}(q, s) = \begin{cases} \{\delta(q, s)\} & \text{se } q \in Q \backslash \{q_1\} \\ \{\widetilde{\delta}(q, s)\} & \text{se } q \in \widetilde{Q} \backslash \{\widetilde{q_1}\} \end{cases}$
  $\check{\delta}(\check{q_1}, s) = \{\delta(q_1, s)\} \cup \{\widetilde{\delta}(\widetilde{q_1}, s)\}$

□

8.4.3 Complementazione

Teorema 8.6 (Chiusura dei linguaggi regolari rispetto alla complementazione)**.** *Se $\mathcal{L}$ è un linguaggio regolare allora $A^* - L$ è un linguaggio regolare.*

Dimostrazione. Dall'ipotesi esiste $\mathscr{M} = \langle A, Q, F, q_1, \delta\rangle$ DFA tale che $\mathcal{L} = L(\mathscr{M})$. Specifichiamo un automa finito $\widehat{\mathscr{M}}$ che riconosca $A^* - L$.

$$\widehat{\mathscr{M}} = \langle A, Q, \widehat{F}, q_1, \delta\rangle, \text{ dove } \widehat{F} = Q \backslash F$$

L'automa $\widehat{\mathscr{M}}$ riconosce $A^* - L$. □

8.4.4 Intersezione

Teorema 8.7 (Chiusura dei linguaggi regolari rispetto all'intersezione)**.** *Se $\mathcal{L}_1$ e $\mathcal{L}_2$ sono linguaggi regolari, allora anche $\mathcal{L}_1 \cap \mathcal{L}_2$ è un linguaggio regolare.*

Dimostrazione. Notiamo innanzitutto che

$$\mathcal{L}_1 \cap \mathcal{L}_2 = A^* - \{\{A^* - \mathcal{L}_1\} \cup \{A^* - \mathcal{L}_2\}\}.$$

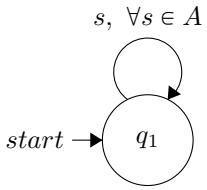
Da ciò e dai teoremi 8.5 e 8.6 segue la tesi. □

8.5 Alcuni linguaggi regolari di base

8.5.1 $\varnothing$ e $\{\varepsilon\}$ sono linguaggi regolari

Teorema 8.8 ($\varnothing$ è un linguaggio regolare)**.** *L'insieme vuoto è un linguaggio regolare.*

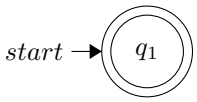
Dimostrazione. Qualsiasi automa privo di stati di accettazione riconosce $\varnothing$. Ad esempio il seguente:



□

Teorema 8.9 ($\{\varepsilon\}$ è un linguaggio regolare)**.** *Il singleton della parola vuota è un linguaggio regolare. Per provarlo, mostriamo un automa che lo riconosca. Posso anche specificare un NDFA, dal momento che abbiamo provato in 8.2 che un linguaggio riconosciuto da un NDFA è regolare.*

Dimostrazione. Ecco un semplice NDFA che riconosce $\{\varepsilon\}$:

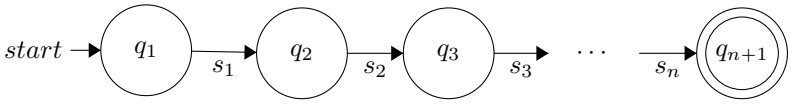


□

8.5.2 $\{u\}$ con $u \in A^*$ è un linguaggio regolare

Teorema 8.10 ($\{u\}$ è un linguaggio regolare)**.** *Sia $u = s_1 \cdots s_k,\ k \in \mathbb{N}$, $\{u\}$ è un linguaggio regolare.*

Dimostrazione. Specifichiamo automa che riconosce $\{u\}$:



□

8.5.3 Tutti i linguaggi finiti sono regolari

Teorema 8.11 (Tutti i linguaggi finiti sono regolari)**.** *Se $L = \{u_1, \cdots, u_k\}$ è un linguaggio finito, allora L è regolare.*

Dimostrazione. Essendo L finito, si avrà $L = \{u_1\} \cup \cdots \cup \{u_k\}$. Quindi, per i teoremi 8.5 e 8.10, L è regolare. □

8.6 Concatenazione

Definizione 8.10 (Concatenazione di due linguaggi)**.** Dati due linguaggi $\mathcal{L}_1$, $\mathcal{L}_2 \subseteq A^*$, si definisce la concatenazione di $\mathcal{L}_1$ con $\mathcal{L}_2$ il linguaggio

$$\mathcal{L}_1 \cdot \mathcal{L}_2 = \{uv \mid u \in \mathcal{L}_1 \wedge v \in \mathcal{L}_2\}$$

Teorema 8.12 ($\mathcal{L}_1, \mathcal{L}_2$ regolari $\Rightarrow \mathcal{L}_1 \cdot \mathcal{L}_2$ regolare)**.** *Se $\mathcal{L}_1$ ed $\mathcal{L}_2$ sono linguaggi regolari, allora la loro concatenazione $\mathcal{L}_1 \cdot \mathcal{L}_2$ è regolare.*

Dimostrazione. Per ipotesi esistono $\mathscr{M}_1 = \langle A, Q_1, F_1, q_1^1, \delta_1\rangle$, $\mathscr{M}_2 = \langle A, Q_2, F_2, q_1^2, \delta_2\rangle$ DFA tali che $L(\mathscr{M}_1) = \mathcal{L}_1$ e $L(\mathscr{M}_2) = \mathcal{L}_2$ e $Q_1 \cap Q_2 = \varnothing$. Dimostriamo la tesi specificando un automa $\mathscr{M}_3 = \langle A, Q_3, F_3, q_1^3, \delta_3\rangle$ che riconosca $\mathcal{L}_1 \cdot \mathcal{L}_2$.

- $Q_3 = Q_1 \cup Q_2$
- $F_3 = \begin{cases} F_2 & \text{se } \varepsilon \notin \mathcal{L}_2 \\ F_1 \cup F_2 & \text{se } \varepsilon \in \mathcal{L}_2 \end{cases}$
- $\delta_3(q,s) = \begin{cases} \{\delta_1(q,s)\} & \text{se } q \in Q_1 - F \\ \{\delta_1(q,s)\} \cup \{\delta_2(q_1^2, s)\} & \text{se } q \in F_1 \\ \{\delta_2(q,s)\} & \text{se } q \in Q_2 \end{cases}$

□

8.7 Operazione *

Definizione 8.11 (L^* di un linguaggio L)**.** Sia $L \subseteq A^*$, si definisce

$$L^* = \{u \mid u = u_1 \cdot u_2 \cdots u_n,\ n \geqslant 0,\ u_i \in L\}$$

Nota. Notiamo che

(i) $\varepsilon \in L^*$ (perchè n può essere anche 0);
(ii) Se L è finito con $L \neq \varnothing$ e $L \neq \{\varepsilon\}$, L^* non è finito.

Teorema 8.13 (Chiusura della classe dei linguaggi regolari rispetto a *)**.** *Se L è un linguaggio regolare, L^* è un linguaggio regolare.*

Dimostrazione. Se L è regolare, allora esiste $\mathscr{M} = \langle A, Q, F, q_1, \delta\rangle$ che lo riconosce. Proviamo la tesi specificando un automa $\widetilde{\mathscr{M}} = \langle \widetilde{A}, \widetilde{Q}, \widetilde{F}, \widetilde{q_1}, \widetilde{\delta}\rangle$ che riconosca L^*:

- $\widetilde{A} = A$;
- $\widetilde{Q} = Q$;
- $\widetilde{F} = \{q_1\}$;
- $\widetilde{q_1} = q_1$;
- $\widetilde{\delta}(q,s) = \begin{cases} \{\delta(q,s)\} & \text{se } \delta(q,s) \notin F \\ \{\delta(q,s)\} \cup \{q_1\} & \text{se } \delta(q,s) \in F \end{cases}$

L'automa $\widetilde{\mathscr{M}}$ così specificato riconosce L^*. □

8.8 Il teorema di Kleene

Teorema 8.14 (Teorema di caratterizzazione degli automi finiti di Kleene)**.**
Un linguaggio è regolare se e solo se può essere ottenuto da linguaggi finiti applicando le operazioni $\cup$, $\cdot$, * *un numero finito di volte.*

Dimostrazione del verso se $\Leftarrow$*.*
Dal teorema 8.11 si ha tutti i linguaggi finiti sono regolari. Dalle chiusure dimostrate nei teoremi 8.5, 8.6 e 8.13 si ha che ogni linguaggio ottenuto da linguaggi finiti applicando $\cup$, $\cdot$, * un numero finito di volte è regolare. □

Dimostrazione del verso solo se $\Rightarrow$*.*
Sia L un linguaggio regolare. Allora esiste $\mathscr{M} = \langle A = \{s_1, \cdots, s_p\}, Q = \{q_1, \cdots, q_n\}, F, q_1, \delta\rangle$ che riconosce L. Definiamo l'insieme delle possibili traiettorie in questo modo:

$R^k_{i,j} = \{x \mid \delta^*(q_i, x) = q_j \wedge \mathscr{M}$ non transita in nessuno stato q_l tale che $l > k$ mentre elabora la parola $x\}$

Procediamo per induzione su k:

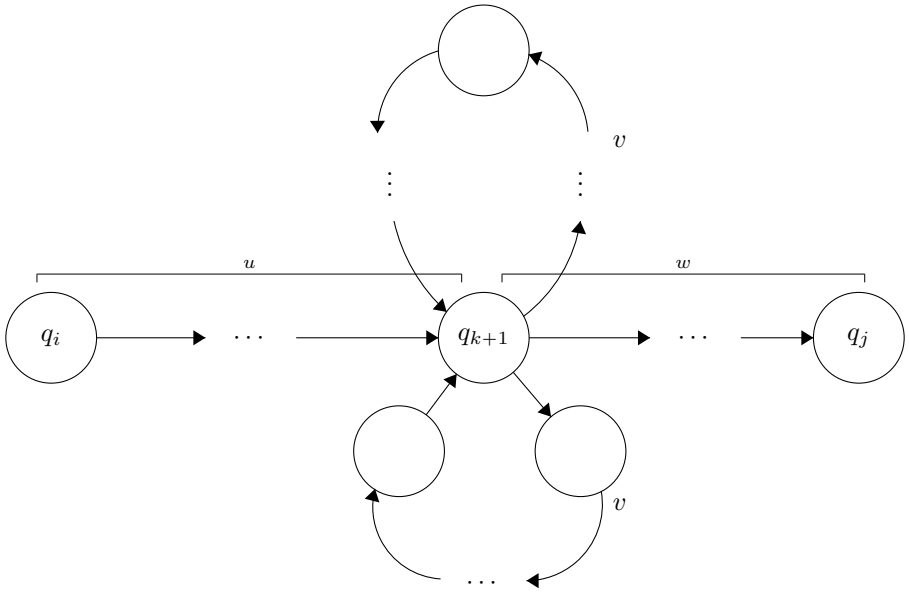
Caso base

$$R^0_{i,j} = \{u \in A^* \mid \delta^*(q_i, u) = q_j \wedge \text{ non si transita per stati intermedi}\}$$

- **Se** $i \neq j$ ci sono al massimo p archi da q_i a q_j. Quindi $R^0_{i,j} \subseteq A^*$ è un insieme finito.
- **Se** $i = j$ allora $R^0_{i,j} = \{\varepsilon\} \subseteq A^*$.
- **Se** $q_i \neq q_j$ e non vi sono archi che li collegano, $R^0_{i,j} = \varnothing$.

In ciascuno dei casi $R^0_{i,j}$ è regolare (può essere ottenuto da linguaggi finiti applicando zero volte le operazioni $\cup, \cdot, ^*$).

Passo induttivo
Per ipotesi induttiva $R^k_{i,j}$ soddisfa la proprietà di essere ottenibile da linguaggi finiti applicando un numero finito di volta le operazioni di $\cup$, $\cdot$, *.
Definiamo $R^{k+1}_{i,j} = R^k_{i,j} \cup \{$percorsi da q_i a q_j che transitano per $q_{k+1}\}$.



Come si può evincere dal grafico, le parole di $R^{k+1}_{i,j}$ avranno forma $x = u \cdot v \cdot w$, dove in particolare $v \in (R^k_{k+1,k+1})^*$, includendo così la parola vuota e loop ripetuti più volte. In generale

$$R^{k+1}_{i,j} = R^k_{i,j} \cup \left[R^k_{i,k+1} \cdot (R^k_{k+1,k+1})^* \cdot R^k_{k+1,j}\right]$$

Provando che anche $R^{k+1}_{i,j}$ soddisfa la proprietà che quindi, per induzione, è valida per ogni $k \geqslant 0$. Infine, dal fatto che

$$L = \bigcup_{q_f \in F} R^n_{1,f},$$

con n numero di stati, segue la tesi. □

8.9 Espressioni regolari

Il teorema di Kleene (8.14) permette di assegnare un nome ad ogni linguaggio regolare in maniera molto semplice. Cominciamo da un alfabeto $A = \{s_1, \cdots, s_k\}$. Quindi definiamo l'alfabeto corrispondente:

$$\widetilde{A} = \{\underline{s_1}, \cdots, \underline{s_k}, \underline{\varepsilon}, \underline{\varnothing}, \underline{\cup}, \underline{\cdot}, \underline{^*}, (,)\}.$$

Definizione 8.12 (Classe delle espressioni regolari su A)**.** La classe delle espressioni regolari sull'alfabeto A è definita come il sottoinsieme di $\widetilde{A}^*$ determinato dalle seguenti:

1. $\underline{\varepsilon}, \underline{\varnothing}, \underline{s_1}, \cdots, \underline{s_k}$ sono espressioni regolari;
2. Se α e β sono espressioni regolari, allora lo è anche $(\alpha \cup \beta)$.
3. Se α e β sono espressioni regolari, allora lo è anche $(\alpha \cdot \beta)$.
4. Se αè un'espressione regolare, allora lo è anche α^*.
5. Nessuna espressione è regolare a meno che non sia generata usando un numero finito di volte le regole 1-4.

Ad ogni espressione regolare γ possiamo associare un corrispondente linguaggio $\langle\gamma\rangle$, determinato secondo le seguenti regole "semantiche" di interpretazione:

$$\begin{aligned}
\langle\underline{s_i}\rangle &= \{s_i\},\\
\langle\underline{\varepsilon}\rangle &= \{\varepsilon\},\\
\langle\underline{\varnothing}\rangle &= \varnothing,\\
\langle(\alpha \cup \beta)\rangle &= \langle\alpha\rangle \cup \langle\beta\rangle,\\
\langle(\alpha \cdot \beta)\rangle &= \langle\alpha\rangle \cdot \langle\beta\rangle,\\
\langle\alpha^*\rangle &= \langle\alpha\rangle^*.
\end{aligned}$$

Teorema 8.15 (Ogni linguaggio finito ha una sua espressione regolare)**.**

Per ogni linguaggio finito $L \subseteq A^*$ esiste un'espressione regolare γ su A tale che $\langle\gamma\rangle = L$.

Dimostrazione. Se $L = \varnothing$, allora $L = \langle\underline{\varnothing}\rangle$. Se $L = \{\varepsilon\}$, allora $L = \langle\underline{\varepsilon}\rangle$. Se $L = \{u\}$, con $u = s_{i_1} \cdots s_{i_k}$, allora $L = \langle(\underline{s_{i_1}} \cdot (\cdots \underline{s_{i_k}}) \cdots)\rangle$. Questo prova la tesi per linguaggi di 0 o 1 elemento. Assumendo (ipotesi induttiva) la tesi vera per linguaggi di k elementi, supponiamo che L abbia $k+1$ elementi. Allora possiamo scrivere

$$L = L_1 \cup \{x\}, \text{ con } x \in A^* \text{ e } |L_1| = k$$

. Per l'ipotesi induttiva esisterà un'espressione regolare α tale che $L_1 = \langle\alpha\rangle$ e, poichè {x} è un linguaggio finito con un elemento, ad esso si applica il caso base mostrato precedentemente ed esisterà l'espressione regolare β tale che $\langle\beta\rangle = \{x\}$. Quindi abbiamo

$$\langle(\alpha \cup \beta)\rangle = \langle\alpha\rangle \cup \langle\beta\rangle = L_1 \cup \{x\} = L,$$

provando il teorema per tutti i linguaggi finiti. □

Teorema 8.16 (Teorema di Kleene - Seconda versione)**.**

Un linguaggio $L \subseteq A^*$ è regolare se e solo se c'è un'espressione regolare γ su A tale che $\langle\gamma\rangle = L$.

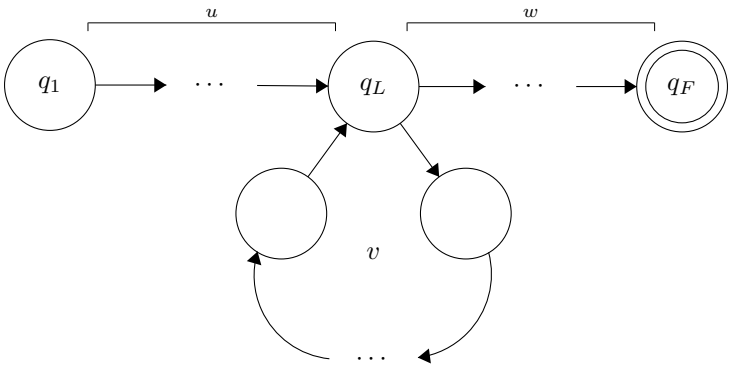
Dimostrazione. Per ogni espressione regolare γ, il linguaggio $\langle\gamma\rangle$ è costruito da linguaggi finiti applicando un numero finito di volte le regole 1-4, cioè le operazioni di $\cup$, $\cdot$,*, e quindi è regolare per il teorema di Kleene (8.14).
Nell'altro verso, sia L un linguaggio regolare. Se L è finito, allora per il teorema 8.15 esisterà l'espressione regolare γ tale che $L = \langle\gamma\rangle$. Altrimenti, per il teorema di Kleene (8.14), L sarà ottenuto da linguaggi finiti applicando un numero finito di volte le operazioni $\cup$, $\cdot$,*. Partendo dai suddetti linguaggi finiti (per i quali esisteranno espressioni regolari), possiamo ottenere un'espressione regolare per L semplicemente indicando ogni uso delle operazioni con il relativo simbolo in $\widetilde{A}$ e usando appropriatamente le parentesi. □

8.10 Pumping Lemma per linguaggi regolari

Lemma 8.17 (Principio della piccionaia)**.** *Se $(n+1)$ oggetti sono distribuiti tra n insiemi, allora almeno uno degli insiemi ne conterrà almeno due.*

Teorema 8.18 (Pumping Lemma)**.** *Sia $L = L(\mathscr{M})$, con $\mathscr{M}_{DFA}$ con n stati. Sia $x \in L$, con $|x| \geqslant n$. Allora possiamo scrivere $x = uvw$ con $v \neq \varepsilon$ e le stringhe $uv^i w \in L\ \forall i \in \mathbb{N}$.*

Dimostrazione. Dal momento che x è composta da n simboli, $\mathscr{M}$ deve effettuare almeno n transizioni di stato mentre analizza x. Sommate allo stato iniziale, otteniamo che $\mathscr{M}$, necessita di almeno $(n+1)$ stati (non necessariamente distinti). Per il principio della piccionaia, possiamo concludere che $\mathscr{M}$ passa almeno una volta per uno stato già visitato. Sia tale stato visitato almeno due volte q_L.



Come si intuisce dal diagramma a bolle, possiamo caratterizzare la computazione dell'automa in questo modo:

$$\delta^*(q_1, u) = q_L$$
$$\delta^*(q_L, v) = q_L$$
$$\delta^*(q_L, w) = q_F \in F$$

Dal momento che il "loop" può essere ripetuto nessuna così come molteplici volte, è chiaro che

$$\delta^*(q_1, uv^i w) = \delta^*(q_1, uvw) \in F,$$

quindi $uv^i w \in L$. □

Corollario 8.18.1. *Se $\mathscr{M}_{DFA}$ con n stati accetta una stringa x tale che $|x| \geqslant n$, allora $L = L(\mathscr{M}_{DFA})$ è infinito.*

Dimostrazione. Conseguenza diretta del teorema 8.18. □

Teorema 8.19.
Sia $\mathscr{M}_{DFA}$ con n stati. Allora, se $L(\mathscr{M}_{DFA}) \neq \varnothing$, esiste una stringa $x \in L(\mathscr{M}_{DFA})$ tale che $|x| < n$.

Dimostrazione. Sia $|x|$ la stringa più breve possibile in $L(\mathscr{M}_{\text{DFA}})$. Supponiamo sia $|x| \geqslant n$. Per il teorema 8.18 (Pumping lemma), $x = uvw$, con $|v| > 0$. Da ciò segue che $|uw| < |x|$, che contraddice l'ipotesi inizale. Dunque $|x| < n$. □

Corollario 8.19.1. *Un $\mathscr{M}_{DFA}$ riconosce $\varnothing \Leftrightarrow \neg\exists x \in L(\mathscr{M}_{DFA})\ :\ |x| < n$*

Questo corollario, conseguenza diretta del teorema 8.19, fornisce un metodo per assicurarsi che un automa riconosca $\varnothing$. Infatti sarà sufficiente testare tutte le stringhe x possibili tali che $|x| < n$. Se nessuna di esse è accettata, allora $L(\mathscr{M}_{\text{DFA}}) = \varnothing$.

Teorema 8.20. *Sia $\mathscr{M}_{DFA}$ con n stati. $L(\mathscr{M}_{DFA})$ è infinito se e solo se $L(\mathscr{M}_{DFA})$ contiene una stringa x tale che $n \leqslant |x| \leqslant 2n$.*

Dimostrazione del verso "se" $\Leftarrow$.
Diretta conseguenza del teorema 8.18 (Pumping lemma). □

Dimostrazione del verso "solo se" $\Rightarrow$.
Se $L(\mathscr{M}_{\text{DFA}})$ è infinito, allora conterrà stringhe di lunghezza $\geqslant 2n$. Sia x la più breve stringa possibile tale che $x \geqslant 2n \wedge x \in L(\mathscr{M}_{\text{DFA}})$. Scriviamo $x = x_1 \cdot x_2$, dove $|x_1| = n$ e $x_2 \geqslant n$. Usando il principio della piccionaia come nella dimostrazione del Pumping lemma (8.18), possiamo scrivere $x_1 = uvw$, con $1 \leqslant |v| \leqslant n$. Dunque $uv^i wx_2 \in L(\mathscr{M}_{\text{DFA}})\ \forall i$, ed in particolare $uwx_2 \in L(\mathscr{M}_{\text{DFA}})$. Ma $|uwx_2| \geqslant |x_2| \geqslant n$ e $|uwx_2| < |x_1x_2| \leqslant 2n$ e, dal momento che x era la più breve parola di lunghezza almeno $2n$, abbiamo $n \leqslant |uwx_2| < 2n$. □

Il teorema fornisce un algorimo per verificare se il linguaggio riconosciuto da un automa $\mathscr{M}$ è infinito. Sarà infatti sufficiente far processare all'automa ciascuna stringa $x\ :\ n \leqslant |x| < 2n$. Se almeno una delle stringhe verrà accettata, allora $L(\mathscr{M})$ è infinito.

Capitolo 9

Grammatiche context-free

9.1 Grammatiche context-free

Definizione 9.1 (Grammatica context-free)**.** Una grammatica context-free (talvolta abbreviato in CF) G è una quadrupla $G = \langle V, \Sigma, R, S \rangle$ dove

V	è un insieme finito detto *alfabeto delle variabili*;
Σ	è un insieme finito detto *alfabeto di terminali*;
$R \neq \varnothing$	è un insieme di regole di forma $X \to u$, con $X \in V, u \in (V \cup \Sigma)^*$;
$S \in V$	è detta variabile iniziale (*assioma*).

Definizione 9.2 (Derivazione unitaria in G)**.** Se $u, v, w \in (V \cup \Sigma)^*$ e $A \to w$ è una regola di G allora la stringa uAv produce *unitariamente* la stringa uwv ($uAv \underset{\text{G}}{\Longrightarrow} uwv$).

Definizione 9.3 (Derivazione in G)**.** $u \overset{*}{\underset{\text{G}}{\Longrightarrow}} v$ è una derivazione in G se

- $u = v$, oppure
- Esiste una successione di stringhe $u_1, \cdots, u_k$ con $k \geqslant 0$ tale che

$$u_1 \underset{\text{G}}{\Longrightarrow} u_2 \underset{\text{G}}{\Longrightarrow} \cdots \underset{\text{G}}{\Longrightarrow} u_k \underset{\text{G}}{\Longrightarrow} v.$$

Definizione 9.4 (Linguaggio generato da G)**.** Il linguaggio generato dalla grammatica G è

$$L(G) = \{u \mid S \overset{*}{\underset{\text{G}}{\Longrightarrow}} u\}$$

Definizione 9.5 (Linguaggio context-free)**.** Un linguaggio si dice context-free (talvolta abbreviato in CF) se e solo se è generato da una grammatica context-free.

9.1.1 Grammatiche regolari

Teorema 9.1 (Tutti i linguaggi regolari sono context-free)**.** *Se L è un linguaggio regolare, allora esiste una grammatica G context-free (regolare) che genera L.*

Dimostrazione. Se L è un linguaggio regolare, allora esisterà $\mathscr{M}$ DFA tale che $L = L(\mathscr{M})$. Dimostriamo la tesi specificando una grammatica $G_{\mathscr{M}}$ tale che $L = L(G_{\mathscr{M}})$.

1. per ogni stato di q_i di $\mathscr{M}$ introduciamo una variabile V_i;
2. Se $\delta(q_i, a) = q_j$ allora si introduce la regola $V_i \to aV_j$;
3. Se $q_i \in F$ si aggiunge la regola $V_i \to \varepsilon$;
4. V_1 è la variabile iniziale ed *assioma* di $G_{\mathscr{M}}$.

□

La grammatica $G_{\mathscr{M}}$ così ottenuta si dice **grammatica regolare**.

Teorema 9.2 (Non tutti i linguaggi CF sono regolari)**.** *Esistono linguaggi CF che non sono regolari.*

Dimostrazione. Si consideri ad esempio il linguaggio $L = \{a^n b^n \mid n > 0\}$. Tale linguaggio è CF in quanto la seguente grammatica lo genera

$$S \to aXb$$
$$X \to aXb|\varepsilon,$$

ma non è regolare in quanto viòla (facile dimostrazione) il pumping lemma. □

9.1.2 Esercizio - mostrare che un linguaggio è CF

Sia $L = \{a^n b^m c^n$ oppure $a^n b^n c^m,\ m, n > 0\}$. Provare che L è un linguaggio CF.

Svolgimento. Specifichiamo una grammatica che riconosca L:

$$S \to aXc$$
$$X \to aXc|bY$$
$$Y \to bY|\varepsilon$$
$$S \to aWbcZ$$
$$W \to aWb|\varepsilon$$
$$Z \to cZ|\varepsilon$$

9.2 Grammatiche in forma normale di Chomsky

Definizione 9.6 (Grammatica in forma normale di Chomsky)**.** Una grammatica context-free si dice in forma normale di Chomsky (in futuro talvolta abbreviato in **CNF** - *Chomsky Normal Form*) se ogni regola ha una delle seguenti forme:

$$A \to BC$$
$$A \to a$$
$$S \to \varepsilon$$

dove $a \in \Sigma$ è un terminale e A, B, C sono variabili qualsiasi tali che B e C non siano la variabile iniziale.

Teorema 9.3 (Ogni linguaggio context-free è generato da una grammatica CNF)**.**
Esiste un algoritmo per trasformare una grammatica context-free G in una grammatica G' CNF in modo tale che $L(G) = L(G')$.

Dimostrazione. Specifichiamo detto algoritmo:

1) Si aggiunge la variabile iniziale S_0 e la regola $S_0 \to S$ (ciò ci garantisce che la variabile iniziale non apparirà sul lato destro di qualche regola.
2) Si eliminano tutte le **produzioni nulle** (ε *regole*) della forma $A \to \varepsilon$ con $A \neq S_0$ e si aggiunge, per ogni occorrenza di A sul lato destro di qualche regola, una nuova regola nella quale quella particolare occorrenza di A è cancellata.
3) Si eliminano le regole unitarie di forma $A \to B$. Se G ha regole della forma $B \to u$ a meno che $A \to u$ sia una regola unitaria già eliminata (u è una stringa di variabili e terminali, cioè $u \in (\Sigma \cup V)^*$).
4) Si sostituisce ogni regola della forma $A \to u_1 \cdots u_k$ con $k \geqslant 3$ e $u_i \in (\Sigma \cup V)$ con le regole:

$$\begin{aligned} A &\to u_1 A_1 \\ A_1 &\to u_2 A_2 \\ &\vdots \\ A_i &\to u_{i+1} A_{i+1} \\ &\vdots \\ A_{k-2} &\to u_{k-1} u_k \end{aligned}$$

Se u_i è un terminale lo si sostituisce con una nuova variabile U_i e si aggiunge la regola $U_i \to u_i$.

□

9.2.1 Esempio di applicazione dell'algoritmo di conversione CNF

Sia data la seguente grammatica G:

$$\begin{aligned} &S \to ASA \mid aB \\ &A \to B \mid S \\ &B \to b \mid \varepsilon. \end{aligned}$$

Specifichiamo una grammatica in forma normale di Chomsky che riconosca $L(G)$.

1) Aggiungiamo una nuova variabile iniziale e la rispettiva regola:

$$\begin{aligned} &S \to ASA \mid aB \\ &A \to B \mid S \\ &B \to b \mid \varepsilon. \end{aligned} \quad \longrightarrow \quad \begin{aligned} &\boldsymbol{S_0 \to S} \\ &S \to ASA \mid aB \\ &A \to B \mid S \\ &B \to b \mid \varepsilon \end{aligned}$$

2) Eliminiamo le produzioni nulle. Nel nostro esempio la produzione nulla è $B \to \varepsilon$, quindi la eliminiamo ed aggiungiamo, per ogni occorrenza di B nella parte destra di una regola, una nuova regola dove l'occorrenza di B è cancellata:

$$\begin{aligned} &S_0 \to S \\ &S \to ASA \mid aB \\ &A \to B \mid S \\ &B \to b \mid \varepsilon. \end{aligned} \quad \longrightarrow \quad \begin{aligned} &S_0 \to S \\ &S \to ASA \mid aB \mid \boldsymbol{a} \\ &A \to B \mid S \mid \boldsymbol{\varepsilon} \\ &B \to b \end{aligned}$$

Quindi eliminiamo anche la $A \to \varepsilon$:

$$\begin{aligned} &S_0 \to S \\ &S \to ASA \mid aB \mid a \\ &A \to B \mid S \mid \varepsilon \\ &B \to b \end{aligned} \quad \longrightarrow \quad \begin{aligned} &S_0 \to S \\ &S \to ASA \mid aB \mid a \mid \boldsymbol{AS} \mid \boldsymbol{SA} \mid \boldsymbol{S} \\ &A \to B \mid S \\ &B \to b \end{aligned}$$

3) Eliminiamo le regole unitarie. Incominciamo eliminando $S \to S$:

$$\begin{aligned} &S_0 \to S \\ &S \to ASA \mid aB \mid a \mid AS \mid SA \mid S \\ &A \to B \mid S \\ &B \to b \end{aligned} \quad \longrightarrow \quad \begin{aligned} &S_0 \to S \\ &S \to \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{AS} \mid \boldsymbol{SA} \\ &A \to B \mid S \\ &B \to b \end{aligned}$$

Quindi eliminiamo $S_0 \to S$:

$$\begin{aligned} &S_0 \to S \\ &S \to ASA \mid aB \mid a \mid AS \mid SA \\ &A \to B \mid S \\ &B \to b \end{aligned} \quad \longrightarrow \quad \begin{aligned} &S_0 \to \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{AS} \mid \boldsymbol{SA} \\ &S \to ASA \mid aB \mid a \mid AS \mid SA \\ &A \to B \mid S \\ &B \to b \end{aligned}$$

Eliminiamo $A \to B$:

$$\begin{aligned} &S_0 \to ASA \mid aB \mid a \mid AS \mid SA \\ &S \to ASA \mid aB \mid a \mid AS \mid SA \\ &A \to B \mid S \\ &B \to b \end{aligned} \quad \longrightarrow \quad \begin{aligned} &S_0 \to ASA \mid aB \mid a \mid AS \mid SA \\ &S \to ASA \mid aB \mid a \mid AS \mid SA \\ &A \to \boldsymbol{b} \mid S \\ &B \to b \end{aligned}$$

Infine eliminiamo $A \to S$:

$$\begin{aligned} &S_0 \to ASA \mid aB \mid a \mid AS \mid SA \\ &S \to ASA \mid aB \mid a \mid AS \mid SA \\ &A \to b \mid S \\ &B \to b \end{aligned} \quad \longrightarrow \quad \begin{aligned} &S_0 \to ASA \mid aB \mid a \mid AS \mid SA \\ &S \to ASA \mid aB \mid a \mid AS \mid SA \\ &A \to b \mid \boldsymbol{ASA} \mid \boldsymbol{aB} \mid \boldsymbol{a} \mid \boldsymbol{AS} \mid \boldsymbol{SA} \\ &B \to b \end{aligned}$$

4) Convertiamo le regole rimaste aggiungendo eventuali ulteriori variabili per soddisfare la forma normale di Chomsky:

$$
\begin{aligned}
S_0 &\to AA_1 \mid U_1B \mid a \mid AS \mid SA \\
A_1 &\to SA \\
U_1 &\to a \\
S &\to AA_1 \mid U_1B \mid a \mid AS \mid SA \\
A &\to b \mid AA_1 \mid U_1B \mid a \mid AS \mid SA \\
B &\to b
\end{aligned}
$$

9.3 Automi a pila

Definizione 9.7 (Automa a pila)**.** Un automa a pila (*pushdown automata*, talvolta abbreviato in PDA) è una sestupla

$$\mathscr{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$$

dove:

1. Q è l'insieme finito degli stati;
2. Σ è un insieme finito di simboli detto *alfabeto del nastro*, con $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$;
3. Γ è un insieme finito di simboli detto *alfabeto della pila*, con $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$;
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ è la funzione di transizione;
5. $q_0 \in Q$ è lo stato iniziale;
6. $F \subseteq Q$ è l'insieme degli stati di accettazione.

Definizione 9.8 (Riconoscimento di una stringa da parte di un automa a pila)**.** L'automa a pila $\mathscr{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ accetta la stringa $w = w_1 \cdot w_2 \cdots w_m$ con $w_i \in \Sigma^*$ se esiste almeno una successione di stati $r_0, \cdots, r_m$ (con $m = |w| + 1$) ed esiste una successione di stringhe $s_0, \cdots, s_m$ con $s_i \in \Gamma^*$ tali che:

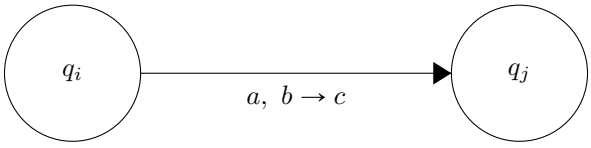
(i) $r_0 = q_0$ e $s_0 = \varepsilon$ (condizioni di inizializzazione).

(ii) $r_m \in F$

(iii) per $0 \leqslant i \leqslant m-1$ si ha che $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ con $s_i = at$ e $s_{i+1} = bt$, $t \in \Gamma^*$, $a, b \in \Gamma_\varepsilon$, $s_0 = \varepsilon$. Questa condizione impone che $\mathscr{M}$ si muova in maniera corretta in base a stato, stack e prossimo simbolo in input.

9.3.1 Rappresentazione di automi a pila

Rappresentazione esemplificativa di una transizione di un automa a pila:

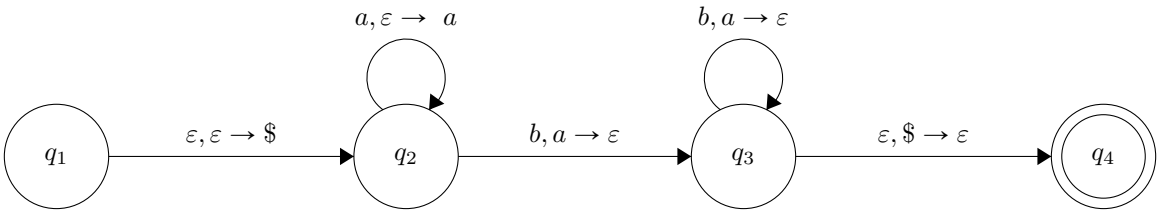


La semantica dietro "$a,\ b \to c$" è la seguente:
l'automa legge a dal nastro (input), legge ed elimina b dallo stack (POP) e scrive quindi c in cima allo stack (PUSH) passando dallo stato q_i allo stato q_j. Alcuni esempi di particolari transizioni sono:

- "$\varepsilon,\ b \to c$": L'automa non legge alcun simbolo dal nastro, legge ed elimina b dallo stack ed effettua il push di c.
- "$a,\ \varepsilon \to c$": Legge a dal nastro e scrive sullo stack c.
- "$\varepsilon,\ \varepsilon \to \varepsilon$": Transita dallo stato q_i allo stato q_j senza compiere altre azioni. Detta anche **transizione spontanea**.

9.3.2 Un automa a pila che riconosce $L = \{a^n b^n \ | n > 0\}$

Mostriamo che gli automi a pila sono capaci di riconoscere linguaggi come $L = \{a^n b^n \ | n > 0\}$ che i semplici DFA non possono riconoscere.



Proviamo a specificarne anche la tabella di transizione

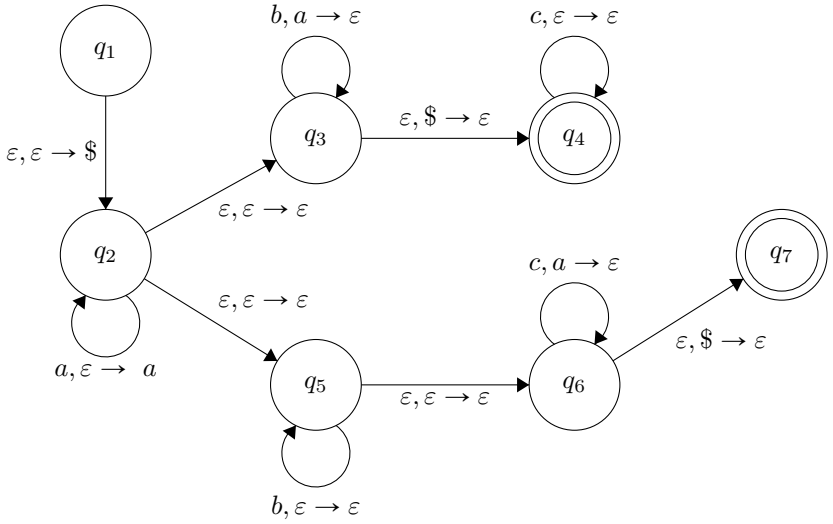
Ingressi	**a**			**b**			ε		
Pila	a	\$	ε	a	\$	ε	a	\$	ε
q_1	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_2, \$\}$
q_2	$\varnothing$	$\varnothing$	$\{q_2, a\}$	$\{q_3, \varepsilon\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
q_3	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3, \varepsilon\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_4, \varepsilon\}$	$\varnothing$
q_4	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Tabella 9.1: Tabella di transizione dell'automa

9.3.3 Un automa a pila che riconosce $L = \{a^i b^j c^k \mid i, j, k \geqslant 0 \ e \ (i = j \ o \ i = k)\}$

Specifichiamo un automa a pila più complesso che riconosca il linguaggio

$$L = \{a^i b^j c^k \mid i, j, k \geqslant 0 \ e \ (i = j \ o \ i = k)\}.$$



9.4 Linguaggi CF e automi a pila

Teorema 9.4 (Se L è CF se allora L è riconosciuto da un automa a pila)**.** *Per ogni linguaggio context-free esiste un automa a pila $\mathscr{M}$ che lo riconosce.*

Dimostrazione. Se un linguaggio L è context-free, allora $\exists\, G_{\text{CF}}$ tale che $L = L(G)$. In base alla grammatica G si specifica in questo modo l'automa a pila $\mathscr{M}$ che riconosce L:

1. **Inizializzazione.** Si scrivono sulla pila il simbolo \$ e il simbolo S di G.
2. **Loop principale.**
   - (2.a) Se in cima alla pila vi è una variabile A, l'automa seleziona **non deterministicamente** una regola di G di forma $A \to u$, cancella A (POP) e scrive sulla pila u, quindi torna al loop principale.
   - (2.b) Se in cima alla pila c'è un terminale a, allora l'automa legge il prossimo simbolo s dal nastro e lo confronta con a, quindi
     - (2.b.i) se $s = a$, cancella a (POP) e torna al loop principale;
     - (2.b.ii) se $a \neq s$, il ramo di computazione termina;
   - (2.c) Se in cima alla pila vi è \$, allora vai nello stato di accettazione.

Prima di descrivere l'automa in maniera formale, mostriamo l'espansione della macro che useremo per scrivere una stringa sullo stack, al fine di semplificare la successiva descrizione dell'automa. Supponiamo di dover scrivere sulla pila la stringa $u = u_1 \cdots u_n$ in corrispondenza della regola $X \to u$ partendo da uno stato q_i entrando in uno stato q_s. Nel farlo possiamo avvalerci dell'insieme di stati di supporto $E_u = \{\overline{q_1}, \cdots, \overline{q_{n-1}}\}$.

$$
\begin{aligned}
(\overline{q_1}, u_l) &\in \delta(q_i, \varepsilon, X) \\
\delta(\overline{q_1}, \varepsilon, \varepsilon) &= \{(\overline{q_2}, u_{n-1})\} \\
\delta(\overline{q_2}, \varepsilon, \varepsilon) &= \{(\overline{q_3}, u_{n-2})\} \\
&\vdots \\
\delta(\overline{q_{n-1}}, \varepsilon, \varepsilon) &= \{(q_f, u_1)\}
\end{aligned}
$$

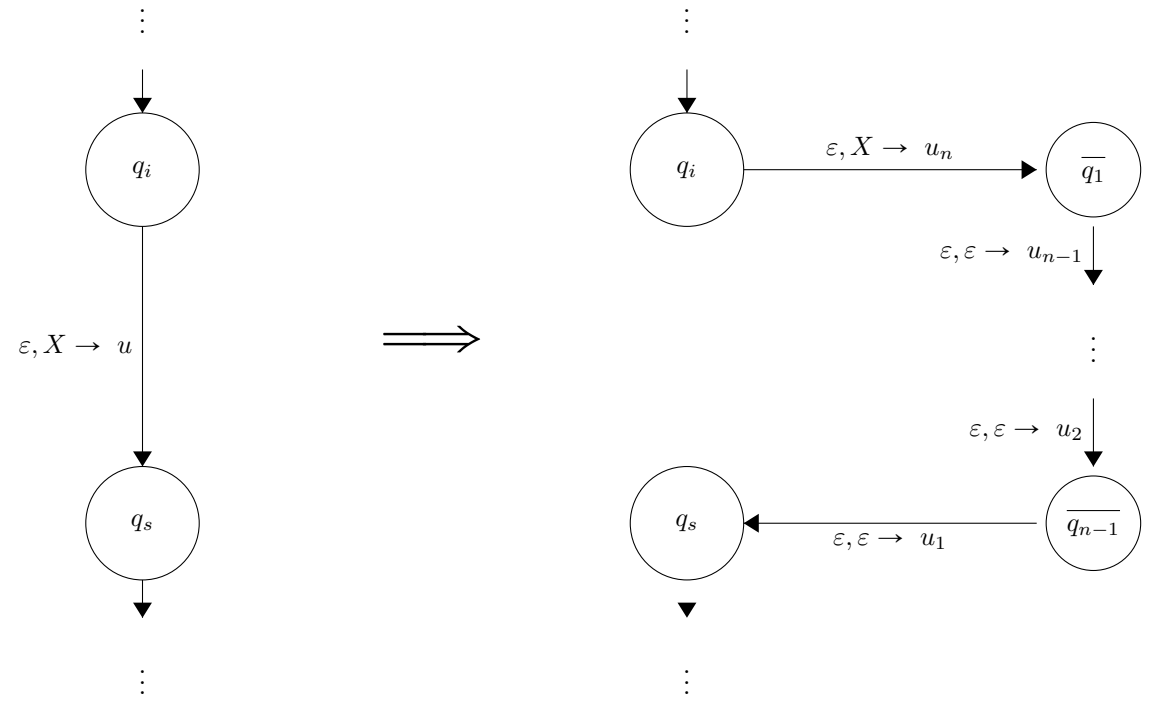
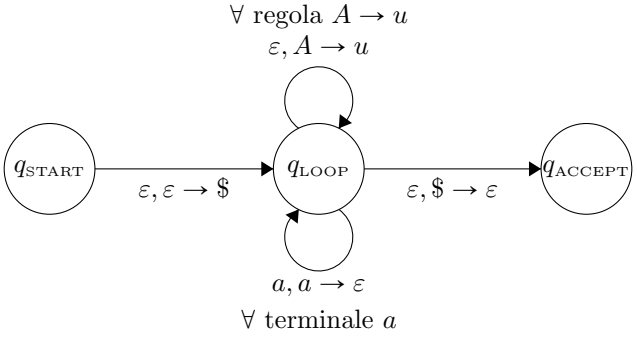


Figura 9.1: Illustrazione della macro (a sinistra) e della sua espansione (a destra).

Specifichiamo infine il PDA $\mathscr{M}$. Avremo $Q = \{q_{\text{START}}, q_{\text{LOOP}}, q_{\text{ACCEPT}}\} \cup E$, con E insieme degli stati di appoggio per la realizzazione delle macro descritte sopra. $F = \{q_{\text{ACCEPT}}\}$.



$$\begin{aligned}
&\delta(q_{\text{START}}, \varepsilon, \varepsilon) = \{(q_{\text{LOOP}}, S\$\} && \textit{(inizializzazione)}\\
&\delta(q_{\text{LOOP}}, \varepsilon, A) = \{(q_{\text{LOOP}}, u) \mid A \to u \text{ è una regola di G}\}\\
&\delta(q_{\text{LOOP}}, a, a) = \{(q_{\text{LOOP}}, \varepsilon)\} && \textit{(per ogni terminale)}\\
&\delta(q_{\text{LOOP}}, \varepsilon, \$) = \{(q_{\text{ACCEPT}}, \varepsilon\}.
\end{aligned}$$

Con la specifica dell'automa $\mathscr{M}$ termina anche la dimostrazione della tesi. □

9.5 Un problema di ambiguità

A volte una grammatica può generare la stessa stringa in modi diversi. Tale stringa avrà differenti alberi di derivazione e dunque differenti significati. Per numerose applicazioni per le quali è necessaria un'interpretazione unica (come ad esempio nel parsing di un codice sorgente in un linguaggio di programmazione) tale evenienza è decisamente sconveniente. Se una grammatica G genera una stringa u in più di un modo, si dice che detta stringa u è *generata ambiguamente* nella grammatica G. Se una grammatica G genera ambiguamente una stringa, allora tale grammatica si dice *ambigua*.

Ad esempio, si consideri la grammatica G:

$$X \to X + X \mid X \cdot X \mid a$$

Questa grammatica genera la stringa $a + a \cdot a$ ambiguamente, infatti:

$$\begin{aligned}
&X \underset{\text{G}}{\Longrightarrow} X \cdot X \underset{\text{G}}{\Longrightarrow} X + X \cdot X \underset{\text{G}}{\Longrightarrow} a + X \cdot X \underset{\text{G}}{\Longrightarrow} a + a \cdot X \underset{\text{G}}{\Longrightarrow} a + a \cdot a\\
&X \underset{\text{G}}{\Longrightarrow} X + X \underset{\text{G}}{\Longrightarrow} a + X \underset{\text{G}}{\Longrightarrow} a + X \cdot X \underset{\text{G}}{\Longrightarrow} a + a \cdot X \underset{\text{G}}{\Longrightarrow} a + a \cdot a
\end{aligned}$$

Formalizzando quanto detto finora, affermiamo che una grammatica genera una stringa ambiguamente se la stringa ha due diversi alberi di derivazione, non due derivazioni distinte. Infatti due derivazioni potrebbero differire semplicemente per l'ordine in cui sono sostituite le variabili ma mantenere lo stesso albero di derivazione. Al fine di concentrarsi sulla struttura, definiamo una regola di derivazione che precisi in che ordine sostituire le variabili.

Definizione 9.9 (Derivazione unitaria estrema a sinistra)**.** Una derivazione unitaria estrema a sinistra da v ad u ($u \underset{\text{L}}{\Longrightarrow} v$) è definita in questo modo: se $u = w_1 X w_2 \underset{\text{L}}{\Longrightarrow} v = w_1 z w_2$, c'è una regola $X \to z$, z *e* $w_2 \in (\Sigma \cup V)^*$, $w_1 \in \Sigma^*$.

Definizione 9.10 (Derivazione estrema a sinistra (DES))**.** Una derivazione estrema a sinistra di v da u ($u \overset{*}{\underset{\text{L}}{\Longrightarrow}} v$) è legittima se esiste una successione si stringhe $u_1, \cdots, u_k$ tali che

$$u \underset{\text{L}}{\Longrightarrow} u_1 \underset{\text{L}}{\Longrightarrow} \cdots \underset{\text{L}}{\Longrightarrow} u_k \underset{\text{L}}{\Longrightarrow} v$$

Le derivazioni presentate nell'esempio precedente di grammatica ambigua sono DES.

Definizione 9.11 (Stringa generata ambiguamente)**.** Una stringa $u \in L(G)$ è generata ambiguamente da G se esistono per u almeno due DES distinte in G.

Definizione 9.12 (Grammatica ambigua)**.** Una grammatica è ambigua se *genera ambiguamente* almeno una stringa $u \in L(G)$.

Definizione 9.13 (Linguaggi inerentemente ambigui)**.** Un linguaggio context-free L è inerentemente ambiguo se e solo se ogni grammatica G che lo genera è ambigua. Equivalentemente, se non esiste una grammatica G non ambigua che lo genera.

Teorema 9.5 (Linguaggi essenzialmente ambigui)**.**
Esistono linguaggi context-free essenzialmente (inerentemente) ambigui.

Un esempio di linguaggio essenzialmente ambiguo è $L = \{a^n b^m c^k \mid n = m \text{ oppure } n = k\}$. È comunque sempre possibile **disambiguare** usando le parentesi.

9.6 Pumping lemma per linguaggi context-free

Teorema 9.6 (Pumping lemma per linguaggi context-free)**.** *Se L è un linguaggio context-free, allora esiste un numero p (detto lunghezza dell'iterazione) tale che, se $w \in L$ e $|w| \geqslant p$, allora $w = uvxyz$ e valgono le seguenti:*

1. $\forall i \geqslant 0$, $uv^i xy^i z \in L$;
2. $|vy| > 0$;
3. $|vxy| \leqslant p$ *(dimostrazione non fatta in classe).*

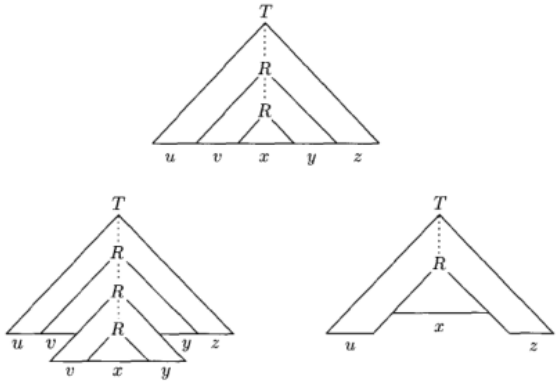
La condizione 2 garantisce che almeno una tra v e y non è la stringa vuota, altrimenti il teorema sarebbe banamente vero.

Dimostrazione. Sia L un linguaggio context-free e sia G la grammatica che lo genera. Sia b il massimo numero di simboli che compare nel lato destro di una regola di G (si potrebbe anche supporre b=2 per il teorema 9.3). In ogni albero di derivazione in questa grammatica, un nodo interno non potrà mai avere più di b figli. Cioè, alpiù b foglie si troveranno a distanza 1 dalla radice dell'albero di derivazione (simbolo iniziale); alpiù b^2 foglie si troveranno a distanza 2; infine alpiù b^h foglie si troveranno a profondità h. Quindi, se l'altezza dell'albero di derivazione è h, la lunghezza della stringa generata sarà alpiù b^h. Dunque una stringa di lunghezza $b^h + 1$ avrà un albero di derivazione di altezza almeno $h + 1$. Definiamo $|V|$ il numero (finito) di variabili nella grammatica G. E assegnamo $p = 2^{|V|} + 1$. Se s è una stringa in L tale che $|s| \geqslant p$, il suo albero di derivazione dovrà avere almeno altezza $|V| + 1$. Sia τ tale albero di derivazione. Se esistono diversi alberi di derivazione, sia τ quello col numero di nodi minimo. Se l'altezza di τ è almeno $|V| + 1$, allora esisterà un percorso dalla radice ad una foglia di lunghezza $|V| + 1$. Tale percorso avrà almeno $|V| + 2$ nodi, e quindi almeno $|V| + 1$ variabili. Per il *principio della piccionaia* (8.17), almeno una variabile comparirà più di una volta nel percorso. Sia R tale variabile che compare almeno due volte (e assumiamo che R sia tra le ultime $|V| + 1$ variabili del percorso). Ogni occorrenza di R ha radicato in sè un sottoalbero. L'occorrenza a profondità inferiore ha radicato il sottoalbero che porterà alle foglie vxy, mentre quella a profondità superiore genererà x.

Entrambi i sottoalberi citati sono radicati nella stessa variabile e, sostituendo un sottoalbero con un altro otteniamo sempre stringhe legittime di $L(G)$. Sostituire il sottoalbero più piccolo con il più grande (*innesto*) genera le stringhe $uv^i xy^i z$, $i > 0$. Al contrario, sostituire il più grande con il più piccolo (*potatura*) genera la stringa $uv^0 xy^0 z$. Dunque il primo punto del teorema è soddisfatto.

Il punto 2 segue dal fatto che v e y non possono essere entrambe nulle. Se lo fossero, infatti, τ non sarebbe l'albero di altezza minima come abbiamo ipotizzato in partenza, perchè l'albero di derivazione ottenuto a seguito della sostituzione del sottoalbero più grande con quello più piccolo genererebbe s ed avrebbe altezza minore.

Il punto 3 segue dalla nostra scelta della variabile R tra quelle che compaiono nelle ultime $|V| + 1$ variabili del percorso. Infatti l'albero radicato nella R più alta potrà avere al massimo altezza $|V| + 2$. Un albero di tale altezza potrà generare al massimo $b^{|V|+2} = p$ foglie. □



9.7 Proprietà di chiusura dei linguaggi context-free

Teorema 9.7 (Linguaggi CF sono chiusi rispetto a $\cup$)**.** *Siano L_1 e L_2 linguaggi CF. Allora $L_1 \cup L_2$ è CF.*

Dimostrazione. Per ipotesi esisteranno $G_1 = \langle V_1, \Sigma_1, R_1, S_1 \rangle$ tale che $L(G_1) = L_1$ e $G_2 = \langle V_2, \Sigma_2, R_2, S_2 \rangle$ tale che $L(G_2) = L_2$. La grammatica $G_3 = \langle V_3, \Sigma_3, R_3, S_3 \rangle$ che riconosce $L_1 \cup L_2$ è così specificata:

- $V_1 \cap V_2 = \varnothing$;
- $V_3 = V_1 \cup V_2 \cup \{S_3\}$;
- $\Sigma_3 = \Sigma_1 \cup \Sigma_2$;
- $R_3 = R_1 \cup R_2 \cup \{S_3 \to S_1 \mid S_2\}$;

□

Teorema 9.8 (Linguaggi CF sono chiusi rispetto a $\cdot$)**.** *Siano L_1 e L_2 linguaggi CF. Allora $L_1 \cdot L_2$ è CF.*

Dimostrazione. Per ipotesi esisteranno $G_1 = \langle V_1, \Sigma_1, R_1, S_1 \rangle$ tale che $L(G_1) = L_1$ e $G_2 = \langle V_2, \Sigma_2, R_2, S_2 \rangle$ tale che $L(G_2) = L_2$. La grammatica $G_3 = \langle V_3, \Sigma_3, R_3, S_3 \rangle$ che riconosce $L_1 \cdot L_2$ è così specificata:

- $V_1 \cap V_2 = \varnothing$;
- $V_3 = V_1 \cup V_2 \cup \{S_3\}$;
- $\Sigma_3 = \Sigma_1 \cup \Sigma_2$;
- $R_3 = R_1 \cup R_2 \cup \{S_3 \to S_1 S_2\}$;

□

Teorema 9.9 (Linguaggi CF sono chiusi rispetto a *)**.** *Sia L_1 un linguaggio CF. Allora L_1^* è CF.*

Dimostrazione. Per ipotesi esisteranno $G_1 = \langle V_1, \Sigma_1, R_1, S_1 \rangle$ tale che $L(G_1) = L_1$. La grammatica $G_3 = \langle V_1, \Sigma_1, R_2, S_1 \rangle$ che riconosce L_1^* è così specificata:

- $R_2 = R_1 \cup \{S_1 \to S_1 S_1 \mid \varepsilon\}$;

□

Teorema 9.10 (Linguaggi CF non sono chiusi rispetto a $\cap$)**.**
Non vale la proprietà di chiusura rispetto all'intersezione. L_1 CF e L_2 CF $\nRightarrow$ $L_1 \cap L_2$ CF.

Dimostrazione per controesempio.

$$L_1 = \{a^n b^n c^j \mid n, j \geqslant 0\}$$
$$L_2 = \{a^n b^j c^n \mid n, j \geqslant 0\}$$
$$L_3 = \{a^n b^n c^n \mid n \geqslant 0\}$$

Si ha che $L_3 = L_1 \cap L_2$, con $L_1,\ L_2$ context-free. Tuttavia L_3 non è context-free perchè viola il pumping lemma! Infatti, supponiamo sia, per assurdo, $L_3 context-free$. Sia p il numero di iterazione per L. La stringa $s = a^p b^p c^p \in L$ e, poichè $|s| > p$, per il pumping lemma (teorema 9.6) $s = uvxyz$, con $|vx| > 0$ e $uv^i xy^i z \in L\ \forall i \geqslant 0$. Analizziamo i possibili casi:

1. Se le stringhe v o y sono composte soltanto da a, soltanto da b o soltanto da c, allora $uv^2xy^2z \notin L$ perchè non ha più il numero corretto di simboli. Assurdo.
2. Se le stringhe v e y contengono simboli diversi tra loro, allora $uv^2xy^2z \notin L$ perchè i simboli non sono più nell'ordine corretto. Assurdo.

Poichè il pumping lemma è valido per tutti i linguggi context-free, ne segue che L non è un linguaggio context-free. □

Teorema 9.11 (Linguaggi CF non sono chiusi rispetto al complemento)**.** *Sia L_1 un linguaggio CF, con Σ alfabeto dei terminali. Allora $\Sigma^* - L_1$ non è necessariamente CF.*

Capitolo 10

La gerarchia di Chomsky

Di seguito risporto la tabella riepilogativa dei linguaggi (e relativi argomenti principali) proposta dal docente.

Processi	Regolari	Context-free	Context-sensitive	Ricorsivi	R.E.	Non R.E.
Riconoscimento	DFA/NDFA.	Automi a pila.	Automi lineari unidirezionali.	S-Progr.	(es. K)	(es. $\overline{K}$)
	Proprietà di chiusura. Teorema di Kleene.	Accettazione, simulazione di grammatiche. Pumping lemma.			NO	NO
Generazione	Grammatiche regolari.	Grammatiche CF.	Grammatiche context-sensitive.	S-Progr.	(es. K).	NO
		Proprietà di chiusura. Derivazione/albero. Ambiguità, Forma normale di Chomsky			S-Prog.	

Tabella 10.1: Tabella riepilogativa

Lista di teoremi

Lista di definizioni