Linguaggi e computabilità

RIASSUNTO DEL LIBRO "AUTOMI, LINGUAGGI E CALCOLABILITÀ [3RD ED]", HOPCROFT

RIASSUNTO DI JACOPO DE ANGELIS

SOMMARIO

OBIETTIVI

L'insegnamento ha l'obiettivo di mettere in relazione elementi della teoria dei linguaggi formali con le basi dell'analisi lessicale e sintattica dei linguaggi di programmazione e di rendere lo studente consapevole dei limiti della computazione. Lo studente sarà in grado di definire grammatiche regolari e libere da contesto che sono necessarie per l'utilizzo di analizzatori sintattici standard

Nota: all'orale potranno essere richieste le seguenti "dimostrazioni". Abbiamo scritto dimostrazioni tra virgolette perché in realtà ci interessano soprattutto i ragionamenti e le idee che stanno dietro, più che i dettagli matematici precisi. Verranno chieste ad esempio le dimostrazioni del fatto che i PDA e i DPDA riconoscono (almeno) tutti i linguaggi regolari, oppure sempre per i PDA la trasformazione da accettazione per stato finale a quella per pila vuota, e viceversa. Oppure, la trasformazione da NFA o da ε-NFA a DFA, o la trasformazione da espressioni regolari a ε-NFA, o da DFA a espressioni regolari, come si riconosce l'intersezione tra due linguaggi regolari (automa prodotto), come si minimizza un DFA , come si dimostra che un linguaggio non è regolare (tramite pumping lemma, e dimostrazione del pumping lemma), come si simula una grammatica context-free con un PDA nondeterministico (quello che ha un solo stato), come funziona la macchina di Turing universale, come funzionano le estensioni e le restrizioni delle macchine di Turing descritte nel capitolo 8, come si codifica in binario una macchina di Turing e come si usa questa codifica nella macchina di Turing universale e per dimostrare che il linguaggio diagonale non è ricorsivamente enumerabile. Infine, sapere dove si possono trovare un linguaggio e il suo complemento nella gerarchia costituita dai linguaggi ricorsivi, ricorsivamente enumerabili e non ricorsivamente enumerabili (e perché).

PROGRAMMA ESTESO

1. Introduzione ai contenuti del corso. I concetti matematici di base per la teoria degli automi
2. Automi a stati finiti deterministici. Automi a stati finiti non deterministici. Un'applicazione: ricerche testuali. Automi a stati finiti con epsilon-transizioni
3. Espressioni regolari. Automi a stati finiti ed espressioni regolari
4. Proprietà dei linguaggi regolari. Pumping Lemma per dimostrare che un linguaggio (non) è regolare. Chiusura di linguaggi regolari rispetto ad operazioni booleane. Equivalenza e minimizzazione di automi
5. Grammatiche. Grammatiche Libere dal Contesto. Alberi sintattici. Applicazioni delle Grammatiche Libere dal Contesto. Ambiguità nelle Grammatiche e nei Linguaggi
6. Macchine di Turing. Problemi che i calcolatori non possono risolvere. La Macchina di Turing. Estensione alla Macchina di Turing semplice. Macchine di Turing ridotte
7. Computabilità. Linguaggi non Ricorsivamente Enumerabili. Linguaggi Ricorsivamente Enumerabile e Ricorsivi. Problemi indecidibili relativi alle Macchine di Turing
8. Analizzatori lessicali e sintattici. Linguaggi di mark-up: XML

CAPITOLI DEL LIBRO

CAPITOLO 1

1.5 I concetti centrali della teoria degli automi

CAPITOLO 2

2.2 Automi a stati finiti deterministici

2.3 Automi a stati finiti non deterministici

2.4 Un'applicazione: ricerche testuali

2.5 Automi a stati finiti con epsilon-transizioni

CAPITOLO 3

3.1 Espressioni regolari

3.2 Automi a stati finiti ed espressioni regolari (tranne paragrafo 3.2.1)

3.3 Applicazioni delle espressioni regolari (lettura consigliata in relazione al corso di Sistemi Operativi)

3.4 Proprietà algebriche per le espressioni regolari (tranne i paragrafi 3.4.6 e 3.4.7)

CAPITOLO 4

4.1 Dimostrare che un linguaggio non è regolare

4.2.1 Chiusura di linguaggi regolari rispetto ad operazioni booleane (in particolare, il prodotto di automi per la chiusura rispetto alla intersezione)

4.4 Equivalenza e minimizzazione di automi

CAPITOLO 5

5.1 Grammatiche Libere dal Contesto

5.2 Alberi sintattici (tranne paragrafi 5.2.4 e 5.2.6)

5.3 Applicazioni delle Grammatiche Libere dal Contesto (lettura consigliata in relazione al corso di Sistemi Operativi)

5.4 Ambiguità nelle Grammatiche e nei Linguaggi

CAPITOLO 6

6.1 Definizione di Automa a Pila (PDA)

6.2 I linguaggi dei PDA

6.3 Equivalenza tra PDA e CFG (tranne paragrafo 6.3.2)

6.4 Automi a Pila Deterministici

CAPITOLO 8

8.1 Problemi che i calcolatori non possono risolvere

8.2 La Macchina di Turing

8.4 Estensione alla Macchina di Turing semplice (tranne il paragrafo 8.4.3)

8.5 Macchine di Turing ridotte (soltanto definizioni e equivalenze con MT semplice)

CAPITOLO 9

9.1 Un linguaggio non Ricorsivamente Enumerabile

9.2 Un problema indecidibile ma Ricorsivamente Enumerabile

9.3 Problemi indecidibili relativi alle Macchine di Turing (cenni al Teorema di Rice, e sue applicazioni)

MATERIALE DIDATTICO

Libro di testo:

- J.E. Hopcroft, R. Motwani, J.D. Ullman, Automi, linguaggi e calcolabilità, Addison Wesley

Materiale fornito sul supporto e-learning

1 AUTOMI: METODO E FOLLIA

La teoria degli automi c lo studio di dispositivi astratti di calcolo, o “macchine”. Negli anni '40 e '50 diversi ricercatori studiarono alcuni tipi più semplici di macchine, che oggi sono dette automi a stati finiti. Nello stesso periodo, nei tardi anni ’50, il linguista Noam Chomsky iniziò a studiare le grammatiche formali.

Nel 1969 S. Cook approfondì gli studi di Turing su ciò che si può calcolare. Cook riuscì a distinguere i problemi risolvibili in modo efficiente da un elaboratore da quelli che possono essere risolti in linea di principio, ma che di fatto richiedono così tanto tempo da rendere inutilizzabile un computer se non per istanzi del problema di dimensione limitata. Questa seconda classe di problemi viene definita “intrattabile” o “NP-hard”.

Tutti questi sviluppi teorici hanno un rapporto diretto con quanto gli informatici fanno attualmente. Alcuni concetti, come gli automi a stati finiti e certi tipi di grammatiche formali, vengono usati nella progettazione e realizzazione di importanti tipi di software. Altri concetti, come la macchina di Turing, aiutano a comprendere che cosa ci si può aspettare dal software.

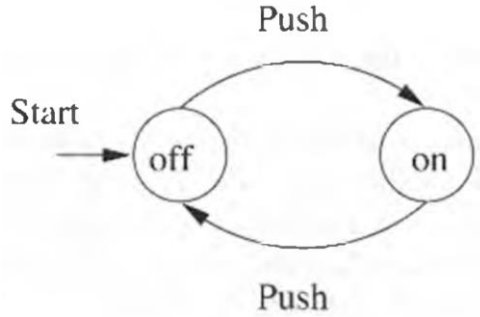
1.1 PERCHÉ STUDIARE LA TEORIA DEGLI AUTOMI

1.1.1 Introduzione agli automi a stati finiti

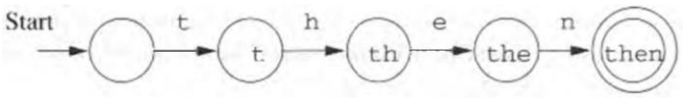
Gli automi a stati finiti sono un utile modello di molte categorie importanti di hardware e software. Alcuni dei casi più significativi:

1) Software per progettare circuiti digitali e verificarne il comportamento
2) L’analizzatore lessicale di un compilatore, ossia il componente che scompone l’input (i dati in ingressi) in unità logiche, come gli identificatori, le parole chiave e la punteggiatura.
3) Software per esaminare vaste collezioni di testi, ad esempio pagine web, per trovare occorrenze di parole o di frasi
4) Software per verificare sistemi di qualsiasi tipo, che abbiamo un numero finito di stati discreti, come i protocolli di comunicazione oppure i protocolli per lo scambio sicuro di informazioni.

Ci sono molti sistemi o componenti, come quelli elencati sopra, di cui si può dire che in ogni istante si trovano in uno “stato” preso da un insieme finito. Uno stato ha lo scopo di ricordare una parte pertinente della storia del sistema. Dal momento che c’è solo un numero finito di stati generalmente non si può ricordare l’intera storia, perciò il sistema dev’essere progettato attentamente affinché ricordi ciò che è importante e dimentichi ciò che non lo è.



Un interruttore è forse il più semplice automa a stati finiti non banale. Come per tutti gli automi a stati finiti, gli stati vengono indicati per mezzo di cerchi: nell’esempio considerato gli stati sono stati chiamati off e on. Gli archi tra gli stati vengono etichettati dagli input, che rappresentano le influenze esterne sul sistema. Qui ambedue gli archi sono etichettati dall’input Push, che rappresenta la pressione del pulsante. Il significato dei due archi è che, indipendentemente dallo stato presente, a fronte di un input Push il sistema passa nell’altro stato. Uno degli stati è detto “stato iniziale”, cioè lo stato in cui il sistema si trova inizialmente. Spesso è necessario indicare uno o più stati come gli stati “finali” o “accettanti”. Per convenzione gli stati accettanti vengono rappresentati da un doppio cerchio.



1.2 I CONCETTI CENTRALI NELLA TEORIA DEGLI AUTOMI

Vedremo ora alcuni dei termini più importanti che permeano la teoria degli automi. Questi concetti sono:

- Alfabeto
- Stringa
- Linguaggio

1.2.1 Alfabeti

Un alfabeto è un insieme finito e non vuoto di simboli.

Per indicarlo si usa il simbolo ∑. Alcuni esempi:

- ∑ = {0, 1}, alfabeto binario
- ∑ = {a, b, …, z}, insieme di tutte le lettere minuscole

1.2.2 Stringhe

Una stringa (o parola) è una sequenza finita di simboli scelti da un alfabeto. Ad esempio "01101" è una stringa dell'alfabeto binario $\sum^1$ = {0, 1}.

STRINGA VUOTA

La stringa vuota è la stringa composta da zero simboli ed è indicata con ε[2].

LUNGHEZZA DI UNA STRINGA

Spesso è utile classificare le stringhe in base alla loro lunghezza, ovvero il numero di posizioni per i simboli della stringa.

Dire che la lunghezza "è il numero di simboli della stringa" è corretto colloquialmente ma sbagliato formalmente poiché 11010 avrebbe lunghezza 2 (0 e 1) in base a questa definizione.

La notazione standard per la lunghezza di una stringa ω[3] è |ω|. Per esempio |001| = 3 e |ε| = 0.

POTENZE DI UN ALFABETO

Se ∑ è un alfabeto possiamo esprimere l'insieme di tutte le stringhe di una certa lunghezza su tale alfabeto usando una notazione esponenziale. Definiamo $\sum^k$ come l'insieme delle stringhe di lunghezza k, con simboli tratti da ∑. Questa notazione rispecchia l'insieme potenza, ovvero l'insieme composto dalle tuple di lunghezza k formato su ∑ x ∑ x … x ∑.

Ad esempio $\sum^3$ = {000, 001, 010, 011, 100, 101, 110, 111} dove ∑ = {0, 1}.

Si evince che il numero di elementi di $\sum^k$ è il numero di simboli dell'alfabeto ∑ elevati alla k.

Si osservi che si crea una certa confusione tra ∑ e $\sum^1$:

- ∑ è un alfabeto i cui membri sono simboli
- $\sum^1$ è un insieme di stringhe, i cui membri sono le stringhe composte dai singoli elementi dell'alfabeto

l'insieme di tutte le stringhe di un alfabeto ∑ viene indicato convenzionalmente con ∑*, formulato altrimenti come $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ldots$.

[1] Epsilon maiuscola
[2] Epsilon minuscola
[3] Omega minuscola

Talvolta si desidera escludere la stringa vuota da un insieme di stringhe. L'insieme delle stringhe non vuote è indicato con $\sum^+$. Di conseguenza valgono due equivalenze, chiamate chiusura di Kleene:

- $\sum^+ = \sum^1 \cup \sum^2 \cup \ldots$.
- $\sum^* = \sum^0 \cup \sum^+$

CONCATENAZIONE DI STRINGHE

Siano x e y stringhe. Allora xy denota la concatenazione di x e y, vale a dire la stringa formata facendo una copia di x e facendola seguire da una copia di y.

1.2.3 Linguaggi

Un insieme di stringhe scelte da $\sum^*$, dove $\sum$ è un particolare alfabeto, si dice un linguaggio. Se $\sum$ è un alfabeto e $L \subseteq \sum^*$, allora L è un linguaggio su $\sum$. Si noti che un linguaggio su $\sum$ non deve necessariamente includere stringhe con tutti i simboli di $\sum$; perciò una volta che abbiamo stabilito che L è un linguaggio su $\sum$, sappiamo anche che è un linguaggio su qualunque alfabeto includa $\sum$.

Nello studio degli automi ci si imbatte in molti altri linguaggi. Alcuni sono esempi astratti. L'unica restrizione di rilievo sui linguaggi è che tutti gli alfabeti sono finiti. Di conseguenza i linguaggi, sebbene possano avere un numero infinito di stringhe, devono consistere di stringhe tratte da un determinato alfabeto finito.

1.2.4 Problemi

Nella teoria degli automi un problema è la questione se una data stringa sia o no membro di un particolare linguaggio. In termini più precisi, se $\sum$ è un alfabeto e L è un linguaggio su $\sum$, allora il problema L è:

- Data una stringa ω in $\sum^*$, decidere se ω appartiene a L.

Un aspetto potenzialmente insoddisfacente della nostra definizione di "problema" è che comunemente non si pensa a un problema in termini di decisione (p è o non è vero?), bensì come se si trattasse di una richiesta di computare o trasformare un certo input (trovare il miglior modo di svolgere un dato compito).

La definizione di problema come linguaggio ha resistito nel tempo come il modo più opportuno di trattare le questioni cruciali della teoria della complessità. In questa teoria l'interesse è volto a scoprire limiti inferiori della complessità di determinati problemi. Di particolare importanza sono le tecniche per dimostrare che certi problemi non possono essere risolti in tempo meno che esponenziale nella dimensione del loro input.

In altre parole, se possiamo dimostrare che è difficile decidere se una data stringa appartiene al linguaggio L_x delle stringhe valide in un linguaggio di programmazione X, allora è evidente che non sarà più facile tradurre in codice oggetto i programmi in linguaggio X: se fosse facile generare il codice, allora si potrebbe eseguire il traduttore e concludere che l'input è un membro valido di L_x nel momento in cui il traduttore riuscisse a produrre il codice oggetto. Dato che il passo finale nel determinare se il codice oggetto è stato prodotto non può essere difficile, si può usare l'algoritmo rapido di generazione del codice oggetto per decidere efficientemente dell'appartenenza a L_x. Così si giunge a contraddire l'assunto che provare l'appartenenza a L_x è difficile. Abbiamo una dimostrazione per assurdo dell'enunciato "se provare l'appartenenza a L_x è difficile, allora compilare programmi nel linguaggio di programmazione X è difficile".

Questa tecnica, cioè mostrare quanto sia complesso un problema usando un suo algoritm,. che si presume efficiente, per risolvere efficientemente un altro problema di cui si conosce già la difficoltà, è detta "riduzione" del secondo problema al primo.

Quindi, per riassumere: un linguaggio è un insieme (eventualmente infinito) di stringhe, formate da simboli tratti dallo stesso alfabeto. Quando le stringhe di un linguaggio devono essere interpretale in qualche modo, la questione se una stringa appartenga al linguaggio viene indicata talora con il termine problema.

2 AUTOMI A STATI FINITI

Questo capitolo presenta la classe dei linguaggi detti "regolari". Tali linguaggi sono esattamente quelli che possono essere descritti dagli automi a stati finiti.

Una distinzione cruciale tra le classi di automi a stati finiti riguarda il controllo:

- Deterministico: l'automa non può essere in più di uno stato per volta
- non deterministico: l'automa può trovarsi in più stati contemporaneamente

Vedremo che l'aggiunta del non determinismo non permette di allargare la classe dei linguaggi definibili da automi a stati finiti deterministici, ma può essere più efficace descrivere un'applicazione usando un automa non deterministico.

2.1 AUTOMI A STATI FINITI DETERMINISTICI (DFA)

2.1.1 Definizione di automa a stati finiti deterministico

Un automa a stati finiti deterministico consiste dei seguenti componenti:

1) Un insieme finito di stati, spesso indicato con Q
2) Un insieme finito di simboli di input, spesso indicato con ∑
3) Una funzione di transizione, che prende come argomento uno stato e un simbolo di input e restituisce uno stato. La funzione di transizione sarà indicata comunemente con δ[4]. Nella rappresentazione grafica informale di automi che abbiamo visto, δ è rappresentata dagli archi tra gli stati e dalle etichette sugli archi. Se q è uno stato e a è un simbolo di input, δ(q, a) è lo stato p tale che esiste un arco etichettato con a da q a p
4) Uno stato iniziale, uno degli stati in Q
5) Un insieme di stati finali, o accettanti, F. l'insieme F è un sottoinsieme di Q

La rappresentazione più concisa di un automa a stati finiti deterministici (DFA) è un'enumerazione dei suoi cinque componenti:

$$\mathbf{A = (Q, \sum, \delta, q_0, F)}$$

Dove:

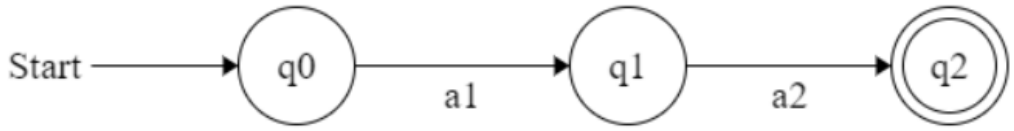
- A è il nome del DFA
- Q è l'insieme degli stati
- ∑ i suoi simboli di input
- δ la sua funzione di transizione
- q_0 il suo stato iniziale
- F il suo insieme di stati accettanti

2.1.2 Elaborazione di stringhe in un DFA

La prima cosa che bisogna capire di un DFA è come decide se "accettare" o no una sequenza di simboli di input. Il "linguaggio" del DFA è l'insieme di tutte le stringhe che il DFA accetta. Supponiamo che $a_1a_2…a_n$ sia una sequenza di simboli di input. Si parte dal DFA nel suo stato inziale, q_0. Consultando la funzione di transizione δ, per esempio $\delta(q_0, a_1) = q_1$, troviamo lo stato in cui il DFA entra dopo aver letto il primo

[4] Delta minuscola

simbolo di input, a_1. Se q_n, ovvero lo stato accettante, è un elemento di F, allora l'input $a_1a_2…a_n$ attraversato per raggiungere q_n, viene accettato, altrimenti viene rifiutato.



ESEMPIO

Specifichiamo formalmente un DFA che accetta tutte e sole le stringhe di 0 e di 1 in cui compare la sequenza 01. Possiamo scrivere il linguaggio L così:

{ω | ω è della forma x01y per stringhe x e y che consistono solamente di 0 e di 1}

Una descrizione equivalente, che usa i parametri x e y a sinistra della barra verticale, è:

{x01y | x e y sono stringhe qualsiasi di 0 e di 1}

Esempi di stringhe appartenenti al linguaggio includono 01, 11010 e 100011. Esempi di stringhe non appartenenti al linguaggio includono ε, 0 e 111000, questo perché la dicitura "x01y" segnala che per appartenere al linguaggio si richiede che tra due stringhe concatenate x e y qualsiasi, deve essere identificabile la sottostringa 01.

Quindi, per decidere se 01 è una sottostringa dell'input, A deve ricordare quanto segue:

1) Hai già visto 01? In caso affermativo accetta ogni sequenza di ulteriori input, cioè da questo momento in poi si troverà solo in stati accettanti
2) Pur non avendo ancora visto 01, l'input più recente è stato 0, cosicché se ora vede un 1 avrà visto 01. Da questo momento può accettare qualunque seguito?
3) Non ha ancora visto 01, ma l'input più recente è nullo (siamo ancora all'inizio) oppure come ultimo dato ha visto un 1? In tal caso A non accetta finché non vede uno 0 e subito dopo un 1

Ognuna di queste tre condizioni può essere rappresentata da uno stato. La condizione 3) è rappresentata da q0. Se si vede un 1 allora non abbiamo fatto alcun passo verso 01, allora $\delta(q_0, 1) = q_0$.

Se invece da q_0 vediamo uno 0, siamo nella condizione 2), ovvero non abbiamo ancora visto un 01 ma abbiamo visto 0, dunque usiamo q_2 per rappresentare la condizione 2), allora $\delta(q_0, 0) = q_2$.

Ora consideriamo la transizione da q_2. Se vediamo uno 0 rimarremo sempre in q_2 poiché non avremo visto 1 ma comunque avremo il primo carattere della sequenza, quindi $\delta(q_2, 0) = q_2$. Nel caso invece vedessimo 1 passeremmo alla condizione 1), ovvero possiamo passare ad uno stato accettante, che si chiamerà q_1 e corrisponderà alla summenzionata condizione 1), ossia $\delta(q_2, 1) = q_1$.

Infine, dobbiamo determinare le transizioni per lo stato 1. In questo caso abbiamo incontrato una sequenza 01, quindi, qualsiasi cosa accada, saremo sempre in una situazione in cui abbiamo visto 01, quindi saremo sempre nello stato accettante q_1, quindi $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.

Da quanto detto risulta allora $Q = \{q_0, q_1, q_2\}$ dove q_0 è lo stato iniziale e l'unico stato accettante è q_1. Quindi $F = \{q_1\}$. La definizione completa dell'automa A che accetta il linguaggio L delle stringhe che hanno una sottostringa 01 è:

$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$

Dove δ è la funzione di transizione descritta sopra.

2.1.3 Notazioni più semplici per i DFA

Un diagramma di transizione per un DFA A = (Q, ∑, δ, q_0, F) è un grafo definito come segue.

- Per ogni stato in Q esiste un nodo
- Per ogni stato q in Q e ogni simbolo di input a in ∑, sia δ(q, a) = p. allora il diagramma ha un arco dal nodo q al nodo p etichettato a. Se esistono diversi simboli di input che causano transizioni da q a p, il diagramma può avere un arco etichettato dalla lista di tali simboli
- Una freccia etichettata Start entra nello stato iniziale q_0. Tale freccia non proviene da alcun nodo.
- I nodi corrispondenti a stati accettanti (quelli in F) sono indicati da un doppio circolo. Gli stati non in F hanno un solo circolo

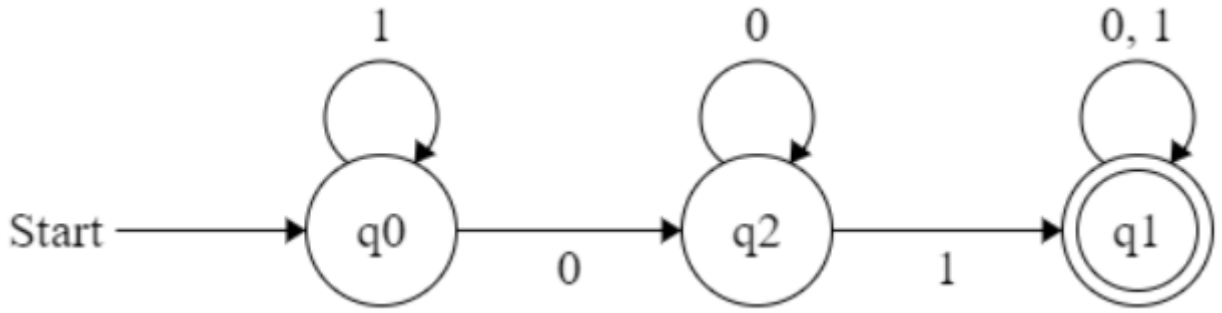


Tabelle di transizione

		0	1
→	q_0	q_2	q_0
*	q_1	q_1	q_1
	q_2	q_2	q_1

Una tabella di transizione è una comune rappresentazione tabellare di una funzione come δ, che ha due argomenti e restituisce un valore.

2.1.4 Estensione della funzione di transizione alle stringhe

Abbiamo spiegato in modo informale che un DFA definisce un linguaggio: l'insieme di tutte le stringhe che producono una sequenza di transizioni di stati dallo stato iniziale a uno stato accettante.

Funzione di transizione estesa $\delta^\wedge$: viene costruita da δ. È una funzione che prende uno stato q e una stringa ω e restituisce uno stato p: lo stato che l'automa raggiunge quando parte nello stato q ed elabora la sequenza di input ω. Definiamo $\delta^\wedge$ per induzione sulla lunghezza della stringa di input come segue:

- Base: $\delta^\wedge(q, \varepsilon) = q$. in altre parole, se ci troviamo nello stato q e non leggiamo alcun input, allora rimaniamo nello stato q e non leggiamo alcun input, allora rimaniamo nello stato q.
- Induzione: supponiamo che ω sia una stringa della forma xa, ossia a è l'ultimo simbolo di ω e x è la stinga che consiste di tutti i simboli eccetto l'ultimo.[5] Per esempio ω = 1101 si scompone in x = 110 e a = 1. Allora
  $\delta^\wedge(q, \omega) = \delta(\delta^\wedge(q, x), a)$
  questa dicitura significa che per computare $\delta^\wedge(q, a)$, calcoliamo prima $\delta^\wedge(q, x)$, lo stato in cui si trova l'automa dopo aver elaborato tutti i simboli di ω eccetto l'ultimo.

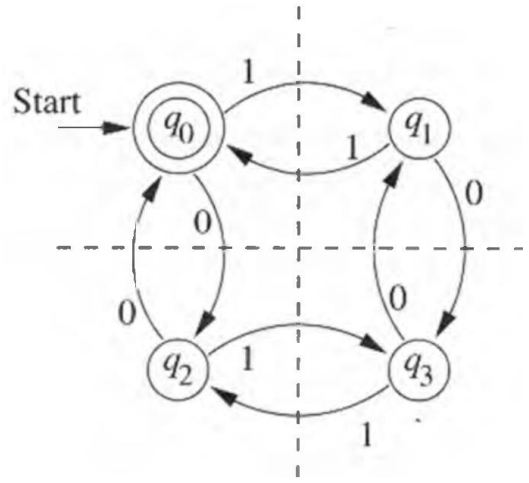
ESEMPIO

Costruiamo un DFA che accetti il linguaggio

[5] Per convenzione: le lettere all'inizio dell'alfabeto sono simboli, quelle verso la fine sono stringhe. Tale convenzione è necessaria affinchè la proposizione "della forma xa" abbia un senso.

$$L = \{\omega \mid \omega \text{ ha un numero pari di 0 e un numero pari di 1}\}$$

In questo caso il numero di 0 e di 1 verrà contato in modulo 2. In altre parole, si usa lo stato per ricordare se il numero degli 0 e il numero degli 1 visti fino a quel momento sono pari o dispari. Quindi ci sono quattro stati, che possono essere interpretati come segue:



- q_0: 0 pari, 1 pari
- q_1: 0 pari, 1 dispari
- q_2: 0 dispari, 1 pari
- q_3: 0 dispari, 1 dispari

lo stato q0 è allo stesso tempo sia stato iniziale che stato accettante.

Siamo ora in grado di specificare un DFA per il linguaggio L:

$$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

La funzione di transizione δ è descritta dal diagramma qua accanto.

		0	1
* →	q_0	q_2	q_1
	q_1	q_3	q_0
	q_2	q_0	q_3
	q_3	q_1	q_2

Ora il nostro intento è usare questo DFA per illustrare la costruzione di $\delta^\wedge$ dalla funzione di transizione δ. Supponiamo che l'input sia 110101, quindi x = 11010 e a = 1.

Avendo numero pari di 0 e di 1 ci aspetteremmo che δ(q0, 110101) = q0.

La verifica comporta il calcolo di δ(q0, ω) per ogni prefisso ω di 110101, a partire da ε e procedendo per aggiunte successive. Il riepilogo di questo calcolo è:

- $\delta^\wedge(q_0, \varepsilon) = q0$
- $\delta^\wedge(q_0, 1) = \delta(\delta^\wedge(q_0, \varepsilon), 1) = \delta(q_0, 1) = q_1$
- $\delta^\wedge(q_0, 11) = \delta(\delta^\wedge(q_0, 1), 1) = \delta(q_1, 1) = q_0$
- $\delta^\wedge(q_0, 110) = \delta(\delta^\wedge(q_0, 11), 0) = \delta(q_0, 0) = q_2$
- $\delta^\wedge(q_0, 1101) = \delta(\delta^\wedge(q_0, 110), 1) = \delta(q_2, 1) = q_3$
- $\delta^\wedge(q_0, 11010) = \delta(\delta^\wedge(q_0, 1101), 0) = \delta(q_3, 0) = q_1$
- $\delta^\wedge(q_0, 110101) = \delta(\delta^\wedge(q_0, 110101), 1) = \delta(q_1, 1) = q_0$

2.1.5 Il linguaggio di un DFA

Possiamo ora definire il linguaggio di un DFA $A = (Q, \sum, \delta, q_0, F)$. Questo linguaggio indicato con L(A) è definito da

$$\mathbf{L(A) = \{\omega \mid \delta^\wedge(q0, \omega) \text{ è in } F\}}$$

In altre parole, il linguaggio A è l'insieme delle stringhe ω che portano dallo stato iniziale q_0 a uno degli stati accettanti. Se L è uguale a L(A) per un DFA A, allora diciamo che L è un "linguaggio regolare".

2.2 AUTOMI A STATI FINITI NON DETERMINISTICI (NFA)

Un automa a stati finiti non deterministico (NFA) può trovarsi contemporaneamente in diversi stati. Questa caratteristica viene spesso espressa come capacità di "scommettere" su certe proprietà dell'input.

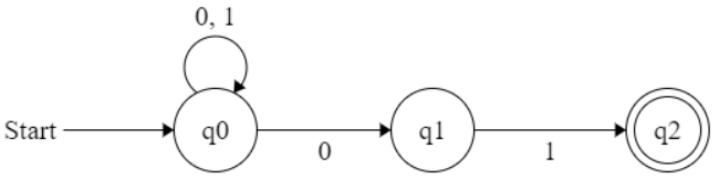
Per esempio, quando un automa viene usato per cercare determinate sequenze di carattere in una lunga porzione di testo, è utile "scommettere" che ci si trova all'inizio di una di tali sequenze e usare una sequenza di stati per verificare, carattere per carattere, che compaia la stinga cercata. Gli NFA accettano

linguaggi regolari. Spesso sono più succinti e più facili da definire rispetto ai DFA, inoltre anche se è sempre possibile convertire un NFA in un DFA, quest'ultimo può avere esponenzialmente più stati di un NFA ma, per fortuna, questi casi sono rari.

2.2.1 Descrizione informale degli automi a stati finiti non deterministici

Come un DFA, un NFA ha un insieme finito di stati, un insieme finito di simboli di input, uno stato iniziale e un insieme di stati accettanti. Ha anche una funzione di transizione che chiameremo δ ma, per gli NFA, δ è una funzione che ha come argomenti uno stato e un simbolo di input ma può restituire zero o più stati.

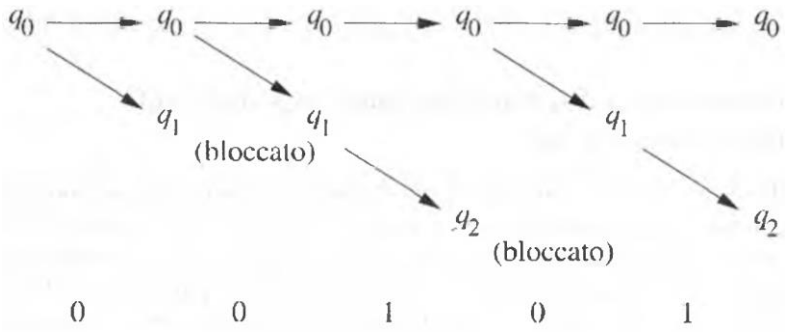
	Input	Output
DFA	Uno stato Un input	Uno stato
NFA	Uno stato Un input	Zero o più stati



Il diagramma qua sopra mostra un NAF con il compito di accettare tutte e sole le stringhe di 0 e di 1 che finiscono per 01. Lo stato q_0 è lo stato iniziale e possiamo pensare che l'automa si trovi nello stato q_0 (eventualmente insieme ad altri stati) quando non ha ancora "scommesso" che il finale 01 è cominciato. È sempre possibile che il simbolo successivo non sia il primo del suffisso 01, anche se quel simbolo è proprio 0. Dunque, lo stato q_0 può operare una transizione verso se stesso sia su 0 sia su 1.

Se però il simbolo successivo è 0, l'NFA scommette anche che è iniziato il suffisso 01. Un altro arco etichettato 0 conduce da q_0 allo stato q_1. Si noti che ci sono due archi etichettati 0 in uscita da q_0. L'NFA ha l'opzione di andare verso q_0 oppure verso q_1, ed effettivamente fa entrambe le cose. Nello stato q_1 l'NFA verifica che il simbolo successivo sia 1, e se è così, passa allo stato q_2 e accetta.

Si osservi che non ci sono archi uscenti da q1 etichettati 0 e non ci sono archi uscenti da q2. In tali situazioni il processo attivo dell'NFA corrispondente a quegli stati viene bloccato. Mentre un DFA ha soltato un unico arco uscente da ogni stato per ogni simbolo di input, un NFA non ha questi vincoli.



la figura qua accanto mostra come un NFA elabora gli input, in questo caso la sequenza è 00101.

2.2.2 Definizione di automa a stati finiti non deterministico

Un NFA si rappresenta essenzialmente come un DFA

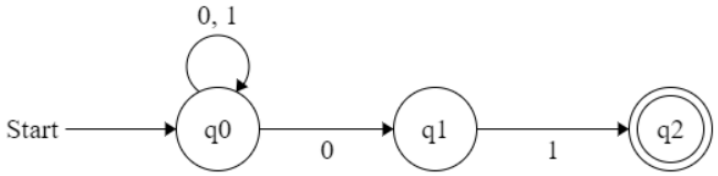
$$\mathbf{A = (Q, \sum, \delta, q_0, F)}$$

Dove:

1) Q è un insieme finito di stati
2) ∑ è un insieme finito di simboli di input
3) q0, elemento di Q, è lo stato iniziale
4) F, un sottoinsieme di Q, è l'insieme degli stati finali (o accettanti)
5) δ, la funzione di transizione, è la funzione che ha come argomenti uno stato in Q e un simbolo di input in ∑, e restituisce un sottoinsieme di Q

quindi l'unica differenza tra DFA e NFA è il tipo di valore restituito da δ: un insieme per un NFA, un singolo dato per un DFA.

Riprendendo l'NFA di prima, può essere rappresentato come:



		0	1
→	q_0	$\{q_0, q_1\}$	$\{q_0\}$
	q_1	Ø	$\{q_2\}$
*	q_2	Ø	Ø

Si osservi che la tabella di transizione funziona sia per un NFA che un DFA, l'unica differenza è che gli NFA prendono come output insiemi.

2.2.3 La funzione di transizione estesa

Come per i DFA, bisogna estendere la transizione δ di un NFA a una funzione δ che prende uno stato q e una stringa di simboli di input ω. In sostanza, se q è l'unico stato nella prima colonna, $\delta^\wedge(q, \delta)$ è la colonna di stati successivi alla lettura di ω.

Per esempio il diagramma sopra mostra che $\delta^\wedge(q0, 001) = \{q0, q_2\}$. In termini formali definiamo $\delta^\wedge$ per la funzione di transizione δ di un NFA come segue:

- base: $\delta^\wedge(q, \varepsilon) = \{q\}$. se nessun simbolo di input è stato letto, ci troviamo nel solo stato da cui siamo partiti.
- Induzione: supponiamo che ω sia della forma ω = xa, dove a è il simbolo finale di ω e x è la parte restante. Supponiamo altresì che $\delta^\wedge(q, x) = \{p_1, p_2, \ldots, p_k\}$ sia

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

  allora $\delta^\wedge(q, \omega) = \{r_1, r_2, \ldots, r_m\}$.

ad esempio, i primi passi per l'input 00101 sono:

- $\delta^\wedge(q, \varepsilon) = \{q_0\}$
- $\delta^\wedge(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
- $\delta^\wedge(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- …

2.2.4 Il linguaggio di un NFA

Il fatto che altre scelte per i simboli di input di ω conducano a uno stato non accettante, oppure ad alcuno stato (cioè che una sequenza di stati muoia), non impedisce a ω di venire accettato dall'NFA nel suo insieme. Formalmente, se A = (Q, ∑, δ, q_0, F) è un NFA, allora

$$\mathbf{L(A) = \{ \omega \mid \delta^\wedge(q0, \omega) \cap F \neq \emptyset\}}$$

In altre parole, L(A) è l'insieme delle stringhe ω in ∑* tali che $\delta^\wedge(q0, \omega)$ contenga almeno uno stato accettante.

2.2.5 Equivalenza di automi a stati finiti deterministici e non deterministici

Ci sono molti linguaggi per i quali è più facile costruire un NFA anziché un DFA, come il linguaggio delle stringhe che finiscono per 01, eppure ogni linguaggio che può essere descritto da un NFA può essere descritto anche da un DFA. Tuttavia, nel peggiore dei casi, il più piccolo DFA può avere 2^n stati, mentre il più piccolo NFA per lo stesso linguaggio ha solo n stati.

In generale, molte dimostrazioni sugli automi richiedono la costruzione di un automa a partire da un altro.

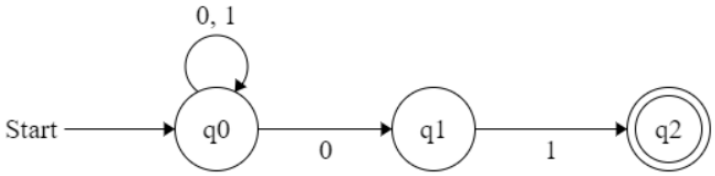
La costruzione per sottoinsiemi parte da un NFA $N = (Q_N, \sum, \delta_N, q_0, F_N)$. il suo fine è la descrizione di un DFA $D = (Q_D, \sum, \delta_D, \{q_0\}, F_D)$ tale che L(D) = L(N). i componenti di D sono costruiti come segue:

- Q_D è l'insieme dei sottoinsiemi di Q_N, vale a dire, Q_D è l'insieme potenza di Q_N. Q_N ha n stati, allora Q_D avrà 2^n stati. Spesso non è possibile raggiungere tutti quegli stati, gli stati inaccessibili possono essere eliminati. Quindi il numero di stati di D può essere molto inferiore a 2^n.
- F_D è l'insieme dei sottoinsiemi S di Q_N tali che $S \cap F_N \neq \emptyset$. In altre parole, F_D è formato dagli insiemi di stati di N che includono almeno uno stato accettante.
- Per ogni insieme $S \subseteq Q_N$ e per ogni simbolo di input a in $\sum$,

$$\delta_D(S, a) \bigcup_{p \text{ in } S} \delta_N(p, a)$$

Cioè, per computare $\delta_D(S, a)$ consideriamo tutti gli stati p in S, rileviamo in quali insiemi di stati l'automa N va a finire partendo da p e leggendo a, e prendiamo infine l'unione di tutti questi insiemi.

ESEMPIO



riprendiamo l'automa precedente che accetta tutte le stringhe che finiscono per 01. Dato che l'insieme di stati di N è $\{q_0, q_1, q_2\}$, la costruzione per sottoinsiemi produce un DFA con 2^3=8 stati, corrispondenti a tutti i sottoinsiemi dei tre stati.

Costruzione completa per sottoinsiemi

		0	1
	ø	ø	ø
→	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
	$\{q_1\}$	ø	$\{q_2\}$
*	$\{q_2\}$	ø	ø
	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
*	$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
*	$\{q_1, q_2\}$	ø	$\{q_2\}$
*	$\{q_1, q_2, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Ridenominazione degli stati

		0	1
	A	A	A
→	B	E	B
	C	A	D
*	D	A	A
	E	E	F
*	F	E	B
*	G	A	D
*	H	E	F

La ridenominazione in questo caso serve per poter definire gli insiemi come singoli stati per un DFA. Possiamo notare che gli stati B, E ed F sono gli unici a poter essere raggiunti.

Gli stati non raggiungibili non devono essere scritti del DFA ridotto.

Teorema: un linguaggio L è accettato da un DFA se e solo se L è accettato da un NFA.

Infatti un DFA è un NFA dove per ogni input c'è al più una sola transizione.a volte però, in situazioni in cui sapiamo che nessuna estensione della sequenza di input può essere accettata, è più opportuno fare in modo che il DFA muoia in un nodo pozzo.

In generale possiamo aggiungere uno stato trappola a qualunque automa che abbia non più di una transizione per ogni stato e simbolo di input: si aggiunge una transizione verso lo stato trappola da uno stato q per tutti i simboli di input per i quali q non ha transizioni.

2.2.6 Un caso sfavorevole di costruzione per sottoinsiemi

nella pratica, è normale che il DFA equivalente abbia approssimativamente lo stesos numero di stati dell'NFA a partire dal quale è stato costruito. Tuttavia c'è una crescita esponensziiale possibile: 2^n stati del DFA a partire da n stati dell'NFA, tutti accessibili.

2.3 UN'APPLICAZIONE: RICERCHE TESTUALI

2.3.1 Automi a stati finiti non deterministici per ricerche testuali

supponiamo che ci venga dato un insieme di parole, che chiameremo parole chiave, e vogliamo cercare le occorrenze di una qualunque di queste parole.

Il testo di un documento viene passato, un carattere alla volta, all'NFA, che vi riconosce le occorrenze delle parole chiave.

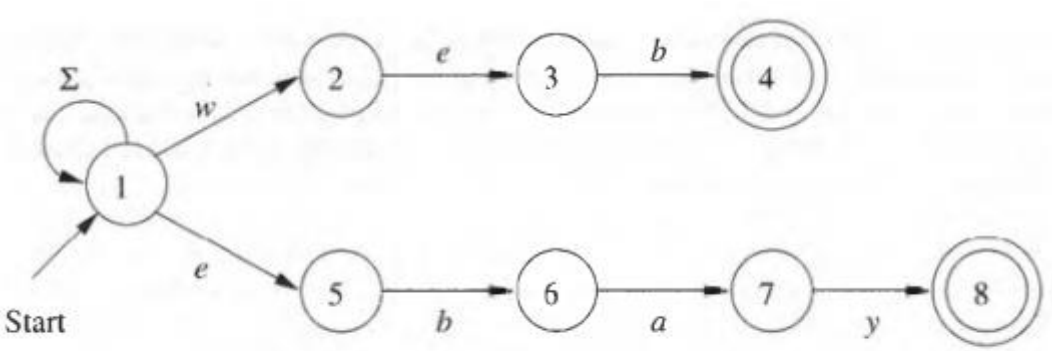


Figura 2.16 Un NFA che cerca le parole web e ebay.

2.4 AUTOMI A STATI FINITI CON EPSILON- TRANSIZIONI

Presentiamo ora un'altra estensione degli automi a stati finiti. La novità consiste nell'ammettere transizioni sulla stringa vuota.

Questa nuova possibilità non amplia la classe dei linguaggi accettati dagli automi a stati finiti, ma offre una certa comodità notazionale.

Vedremo come gli NFA con epsilon transizioni, che chiameremo epsilon-NFA (ε-NFA), sono strettamente legati alle espressioni regolari.

2.4.1 Uso delle epsilon-transizioni

Negli esempi che seguono, gli automi accettano le sequenze di etichette lungo cammini dello stato iniziale a uno stato accettante, considerando come invisibili le occorrenze di ε.

2.4.2 La notazione formale per gli ε-NFA

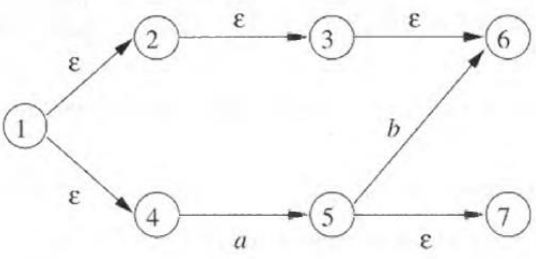
Possiamo rappresentare gli ε-NFA esattamente come un NFA, salvo che per un aspetto: la funzioen di transizione deve incorporare informazioni sulle transizioni etichettate da ε.

Formalmente denotiamo un ε-NFA con

A = (Q, ∑, δ, q0, F)

- Q: insieme degli stati
- ∑: alfabeto dell'NFA, unito a ε
- δ: insieme delle transizioni
- q0: stato inziale
- F: insieme degli stati accettanti

2.4.3 Epsilon chiusure



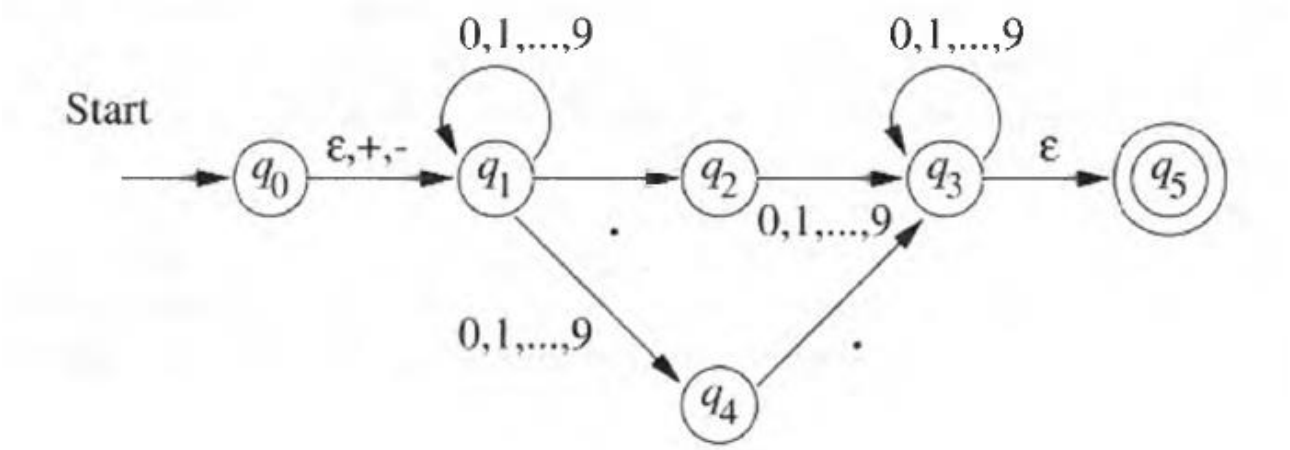
Per ε-chiudere uno stato q si seguono tutte le ε-transizioni uscenti da q, ripetendo poi l'operazione da tutti gli stati raggiunti via via, fino a trovare tutti gli stati raggiungibili da q attraverso cammini solo etichettati come ε-transizioni.

La ε-chiusura viene chiamata ECLOSE(q).

Nell'esempio qua accanto ECLOSE(1) = {1, 2, 3, 4, 6}.

2.4.4 Transizioni estese e linguaggi per gli ε-NFA

Ora che abbiamo definito come vengono dichiarate le ECLOSE, possiamo anche capire come si comporta un ε-NFA a fronte di una sequenza di input diversi da ε.



Ad esempio, calcoliamo δ(q0, 5.) per l' ε-NFA qua sopra:

- Prima definiamo l'eclose dello stato: ECLOSE(q0) = {q0, q1}
- Poi si calcola in sequenza le transizioni per ogni carattere:
  - ECLOSE(q0, 5) = δ(q0, 5) U δ(q1, 5) = {q1, q4}
  - Quindi si applica la ECLOSE ai due stati: ECLOSE(q1) U ECLOSE(q4) = {q1} U {q4} = {q1, q4}
- Poi si continua con la stringa:
  - δ^(q0, 5.) = δ((δ(q1, 5) U δ (q0, 5)), .) = δ(q1, .) U δ(q4, .) = {q2} U {q3} = {q2, q3}
  - ECLOSE(q0, 5.) = ECLOSE(q2) U ECLOSE(q3) = {q2} U {q3, q5} = {q2, q3, q5}

Quindi, in ordine:

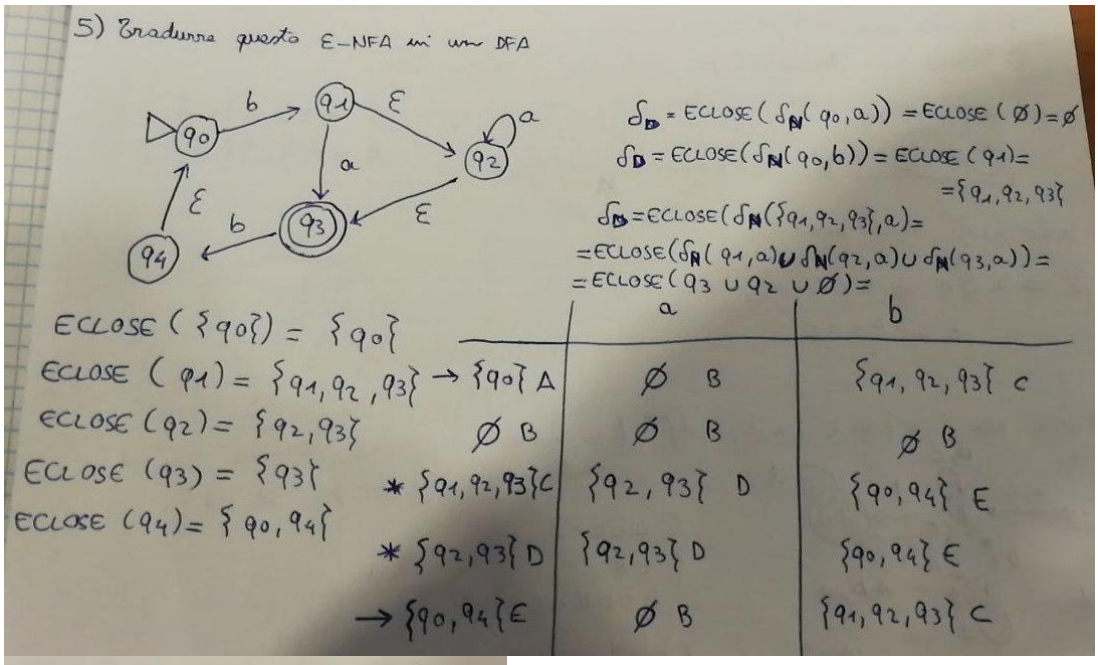
- Prima calcolo le eclose iniziali
- Poi controllo dove può andare per ogni stato l'eclose in base al primo input
- Poi per ogni input calcolato prima uso lo stesso sistema e vedo dove può andare

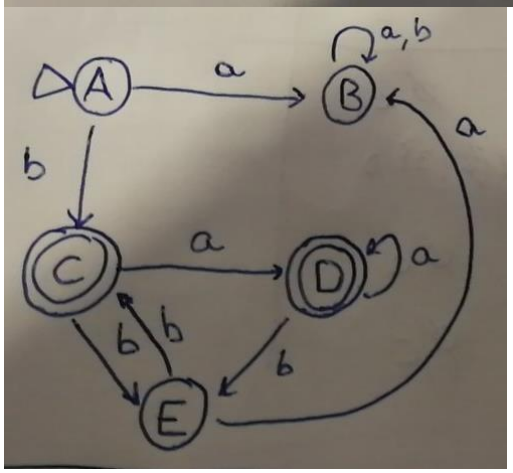
2.4.5 Eliminazione di ε -transizioni

Per eliminare le ε-transizioni, possiamo transformare l' ε-NFA E in un DFA D che accetta lo stesso linguaggio.

I passi sono:

- Prima troviamo tutte le eclose e identifichiamo quelle che contengono lo stato iniziale o uno stato accettante
- Iniziamo dalla prima eclose con stato iniziale e controlliamo dove può andare in base all'input
- In base alle eclose che risultano dalla transizione precedente, mettiamole nella tabella e continuiamo così fino a quando possiamo
- In caso un input non sia elencato, creare lo stato pozzo
- Dare un nome alle varie eclose
- Riscrivere l' ε-NFA in forma di DFA dove i vari stati sono le eclose elencate prima





3 ESPRESSIONI E LINGUAGGI REGOLARI

3.1 ESPRESSIONI REGOLARI

le espressioni regolari offrono qualcosa in più rispetto agli automi: un modo dichiarativo di esprimere le stringhe da accettre.

3.1.1 Gli operatori delle espressioni regolari (ER)

le espressioni regolari denotano linguaggi. Prima di descrivere la notazione delle ER, dobbiamo conoscere le tre operazioni sui linguaggi rappresentate dagli operatori delle ER:

- i linguaggi possono essere uniti
  - L = {a, b, c}, M = {c, d, e}-→L U M = {a, b, c, d, e}
- la oncatenzione di due linguaggi avviene unendo tutte le stringhe del primo linguaggio con tutte quelle del secondo, creando un insieme di dimensione n*m, dove n ed m sono le dimensioni dei linguaggi coinvolti
- la chiusura (o star o chiusura di Kleene) di un linguaggio L viene indicaa con L* e rappresenta l'insieme delle stringhe che possono essere formate con gli elementi dell'alfabeto di L. la differenza tra Σ e Σ* è che in Σ* viene compresa anche la stringa vuota.

3.1.2 Costruzione di espressioni regolari

per riassumere in maniera semplice come vengono costruite le espressioni:

- (A + B): l'opertore + indica la scelta di uno dei due elementi, o A o B
- A*: l'operatore * indica che l'elemento associato può essere scelto un numero arbitrario di volte, anche nessuna

esempio: (01)* + ((10)* + 0(10)*)

quindi, i primi due blocchi sono (01)* + ((10)* + 0(10)*), quidi la scelta è tra scrivere un numero arbitrario di volte 01 o scrivere una volta il contenuto della seconda parentesi

nella seconda parentesi vediamo: (10)* + 0(10)*, ovvero o scrivere un numero arbitrario di volte 10 oppure scrivere 0 e poi concatenato un numero arbitrario di volte 10.

3.1.3 Precedenza degli operatori nelle ER

1) *
2) concatenazione
3) +

3.2 AUTOMI A STATI FINITI ED ESPRESSIONI REGOLARI

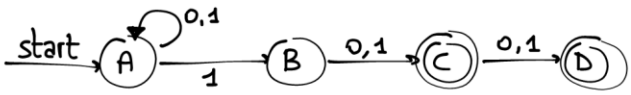
Per provare che le espressioni regolari definiscono la stessa classe, dobbiamo dimostrare due cose:

- ogni linguaggio definito da uno di questi automi è definito anche da un'espressione regolare.
  - Per questa dimostrazione possiamo assumere che il linguaggio sia accettato da un DFA
- ogni linguaggio definito da un'espressione regolare è definito da uno di questi automi.
  - Per questa parte della dimostrazione basta mostrare che esite un NFA con epsilon-transizioni che accetta lo stesso linguaggio

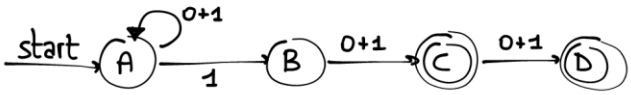
3.2.1 Conversione di DFA in espressioni regolari per eliminazione di stati

dal DFA possiamo ricavare l'ER tramite la riduzione degli stati, ovvero tramite l'unione delle transizioni. Per farlo capire ecco un esempio pratico:

Esempio (3.6 pag 94)
Da DFA (o NFA) a ER



Tale automa accetta le stringhe binarie che hanno un uno in penultima o terzultima posizione.
Rietichettiamo con le ER

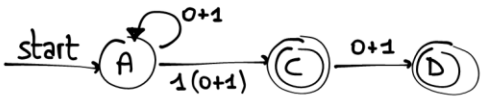


Eliminiamo B, che non è né iniziale né finale. Il predecessore è A e il successore è C.

pred:
A $A \to B : 1$
succ:
C $B \to C : 0+1$
no self loop: $\varnothing$
$R_{A,C}$: $\varnothing$

Diventa $R_{A,C} = \varnothing + 1\varnothing^{*}(0+1) = 1(0+1)$

L'automa è ora



Ci sono ora solo stati iniziali e finali. L'espressione cercata sarà quindi E = E1 + E2.

Eliminiamo C

pred:
A $A \to C : 1(0+1)$

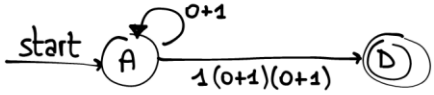
succ:
D $C \to D : 0+1$

no self loop: $\varnothing$

$R_{A,D}$: $\varnothing$

Diventa $R_{A,D} = \varnothing + 1(0+1)\varnothing^*(0+1) = 1(0+1)(0+1)$.

Otteniamo



quindi $E_1 = ((0+1) + 1(0+1)(0+1)\varnothing^*\varnothing)^* 1(0+1)(0+1)\varnothing^*$
$= (0+1)^* 1(0+1)(0+1)$

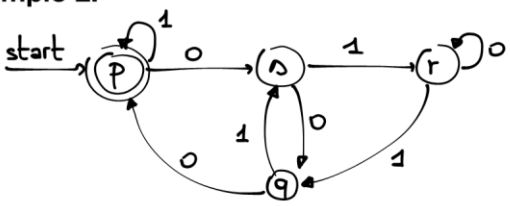
Eliminiamo D, che non ha successori. Quindi

$E_2 = (0+1)^* 1(0+1)$

L'espressione finale è:

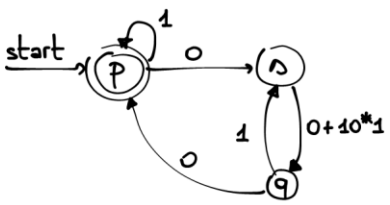
$E = E_1 + E_2 = (0+1)^* 1(0+1)(0+1) + (0+1)^* 1(0+1)$
$= (0+1)^* 1(0+1)(\varepsilon + 0 + 1)$

Esempio 2.



Eliminazione r

pred: s $s \to r : 1$
succ: q $r \to q : 1$
self loop: 0
$R_{s,q}$: 0
Diventa $R_{s,q} = 0 + 10^*1$



Eliminazione s

pred: p, q $p \to s: 0$, $q \to s: 1$

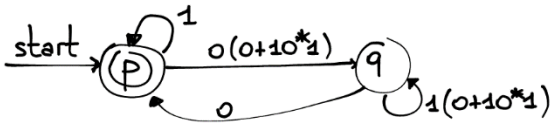
succ: q $s \to q: 0+10^*1$

no self loop: $\varnothing$

$R_{p,q} = \varnothing$, $R_{q,q} = \varnothing$

Diventa $R_{p,q} = \varnothing + 0\varnothing^*(0+10^*1) = 0(0+10^*1)$

$R_{q,q} = \varnothing + 10^*(0+10^*1) = 1(0+10^*1)$



Eliminazione q

pred: p $p \to q: 0(0+10^*1)$

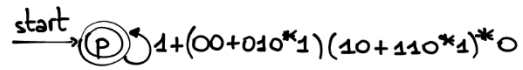
succ: p $q \to p: 0$

self loop: $1(0+10^*1)$

$R_{p,p} = 1$

Diventa $R_{p,p} = 1 + 0(0+10^*1)(1(0+10^*1))^*0$

$= 1+(00+010^*1)(10+110^*1)^*0$



p è iniziale e finale, quindi

$E = (1+(00+010^*1)(10+110^*1)^*0)^*$

Chiusura di un linguaggio regolare

Sia REG la classe dei linguaggi regolari, ovvero qualunque linguaggio regolare su qualunque alfabeto finito non vuoto.

$L, M \in REG \longrightarrow L \cup M \in REG$? (Sì)

Dim 1 (via exp reg)

$L, M \in REG \Rightarrow \exists R, S$ espressioni regolari tali che

$L(R) = L$ e $L(S) = M$

$L \cup M = L(R+S)$ e quindi appartiene a REG

ogni volta che eliminiamo uno stato s, tutti i cammini che passano per s non esistono più per l'automa. Per non cambiare il linguaggio dell'automa dobbiamo includere, su un arco che va direttmente da p a q, le etichette di cammini che vanno da q a p attraverso s.

arriverem dunque a considerare automi che hanno come etichette espressioni regolari. Il linguaggio dell'automa è l'unione, su tutti i cammini dallo stato iniziale a uno stato accettante, dei linguaggi formati concatenando i linguaggi delle ER lungo un cammino.

3.2.2 Conversione di espressioni regolari in automi

Mostriamo come si "passa" dalle ER agli epsilon NFA. L'idea è sfruttare la definizione ricorsiva delle ER.
Una ER è:

base: ε, $\varnothing$, $a\ (\in \Sigma)$

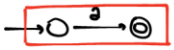
passo: $S+T$, ST, S^*, (S)

Costruiamo dei "moduli" per ciascun caso.
Casi base:

1. $R=\varepsilon$ $\quad L(R)=L(\varepsilon)=\varepsilon$

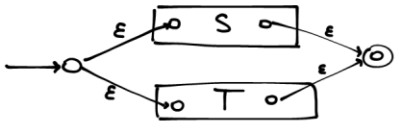
2. $R=\varnothing$ $\quad L(R)=L(\varnothing)=\varnothing$

3. $R=a\ (\in\Sigma)$ $\quad L(R)=L(a)=\{a\}$

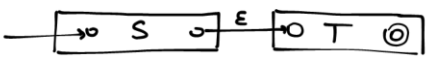


Passo:

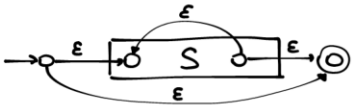
1. $R=S+T$ $\quad L(R)=L(S)\cup L(T)$



2. $R=ST$ $\quad L(R)=L(S)L(T)$



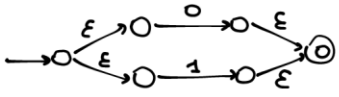
3. $R=S^*$ $\quad L(R)=(L(S))^*$



4. $R=(S)$ $\quad L(R)=L(S)$

Mostriamo un esempio dall'ER $(0+1)*1(0+1)$

- ε-NFA per $0+1$



tramite i "blocchi" descritti qua sopra possiamo trasformare una qualsiasi espressione regolare in un automa che sfrutta le epsilon-transizioni.

[[inserire altri esempi]]

3.3 PROPRIETÀ ALGEBRICHE PER LE ESPRESSIONI REGOLARI

3.3.1 Associatività e commutatività

Dati i linguaggi L, M, N:

- Commutatività: L + M = M + L, l'unione di due linguaggi è effettuabile in qualsiasi ordine
- Associatività per l'unione: (L + M) + N = L + (M + N)
- Associatività per la concatenazione: (LM)N = L(MN)

3.3.2 Proprietà distributive

Una proprietà distributiva coinvolge due operatori e afferma che un operatore può essere applicato a ogni argomento dell'altro operatore individualmente.

- L(M + N) = LM + LN
- (M + N)L = ML + NL

3.3.3 Proprietà relative alla chiusura

- $(L^*)^* = L^*$
- $Ø^* = ε$, la chiusura del linguaggio vuoto produce un linguaggio con la sola stringa vuota
- $ε^* = ε$
- $L^+ = L + LL + LLL + …$
- $L^+ = LL^*$
- $L^* = L^+ + ε$
- $L? = ε + L$

4 PROPRIETÀ DEI LINGUAGGI REGOLARI

4.1 DIMOSTRARE CHE UN LINGUAGGIO NON È REGOLARE

per dimostrare che certi linguaggi non lo sono, introduciamo in questo paragrafo una tecnica efficace, nota come pumping lemma.

4.1.1 Il pumping lemma per i linguaggi regolari

teorema: sia L un linguaggio regolare. Allora esiste una costante n (dipendente da L) tale che, per ogni stringa w in L per la quale |w| >= n, possiamo scomporre w in tre stringhe w = xyz in modo che:

- y != epsilon
- |xy| <= n
- per ogni k >= 0 anche la stringa xy^kz è in L

in altre parole possiamo trovare una stringa non vuota y che possiamo ripetere quante volte vogliamo o anche cancellare senza uscire dal linguaggio.

La dimostrazione consiste nel visualizzare un PDA con tre stati adibiti a x, y e z.

ogni stringa di lunghezza superiore al numero di stati obbliga a ripetere uno stato. Quindi per ogni input superiore al numero di stati ci troviamo costretti a riutilizzare uno stato già usato, quindi cicleremo su di uno degli stati.

4.1.2 Applicazioni del pumping lemma

prendete il linguaggio a^nb^n

dovremmo trovare una divisione tale che la parte centrale possa essere replicata un numero indefinito di volte senza uscire dal linguaggio affinchè sia regolare.

Proviamo a vedere i tre casi:

1) la parte di ripetizione è tutta tra le a: aaaabbbb
2) la parte di ripetizione è tutta tra le b: aaaabbbb
3) la parte di ripetizione è in mezzo: aaaabbbb

in tutti e tre i casi provando a includere più volte la parte evidenziata usciremmo dal linguaggio, quindi non è regolare.

5 GRAMMATICHE E LINGUAGGI LIBERI DAL CONTESTO

I linguaggi liberi dal contesto sono linguaggi che hanno una notazione naturale ricorsiva, chiamata "grammatiche libere dal contesto" (Contex Free Grammars – CFG). Grazie ad essere la realizzazione di un parser (una funzione che estrae la struttura di un programma) è passata da un'attività di implementazione ad hoc a un lavoro di routine.

Recentemente le CFG sono state usate per descrivere formati di documenti attraverso le cosiddette DTD (Document Type Definition), utilizzate dagli utenti di XML (eXtensible Markup Language) per lo scambio di informazioni nel web.

5.1 GRAMMATICHE LIBERE DAL CONTESTO

5.1.1 Un esempio informale

Consideriamo il linguaggio delle palindrome, ovvero stringhe che si possono leggere da sinistra a destra e al contrario. In altri termini una stringa è palindroma se e solo se $\omega = \omega^R$. Per semplificare le cose descriveremo solo le stringhe palindrome sull'alfabeto {0, 1}.

È facile verificare che il linguaggio L_{pal} delle palindrome di 0 e 1 non è un linguaggio regolare. Per farlo ci serviamo del "pumping lemma". Se L_{pal} è regolare, sia n la costante associata, e consideriamo la stringa palindroma $\omega = 0^n10^n$. per il lemma possiamo scomporre ω in ω = xyz, in modo tale che y consista di uno o più 0 dal primo gruppo. Di conseguenza xz, che dovrebbe trovarsi in L_{pal} se L_{pal} fosse regolare, avrebbe meno 0 a sinistra dell'unico 1 rispetto a quelli a destra, dunque xz non può essere palindroma. Abbiamo così confutato l'ipotesi che L_{pal} sia un linguaggio regolare.

Per stabilire in quali casi una stringa di 0 e 1 si trova in L_{pal}, possiamo avvalerci di una semplice definizione ricorsiva. La base della definizione stabilisce che alcune stringhe semplici si trovano in L_{pal}, si sfrutta poi il fatto che se una stringa è palindroma deve cominciare e finire con lo stesso simbolo. Inoltre, quando il primo e l'ultimo simbolo vengono rimossi, la stringa risultante deve essere palindroma.

- **Base**: ε, 0 e 1 sono palindrome
- **Induzione**: se ω è palindroma, lo sono anche 0ω0 e 1ω1. Nessuna stringa di 0 e 1 è palindroma salvo che non risulti dalla base e dalla regola di induzione appena esposte.

Una CFG è una notazione formale per esprimere simili definizioni ricorsive di linguaggi. Una grammatica consiste di una o più variabili che rappresentano classi di stringhe, ossia linguaggi.

Nell'esempio ci occorre una sola variabile, P, che rappresenta l'insieme delle palindrome, cioè la classe delle stringhe che formano il linguaggio L_{pal}.

ESEMPIO

1.	P	→	ε
2.	P	→	0
3.	P	→	1
4.	P	→	0ω0
5.	P	→	1ω1

Le regole che definiscono le palindrome, espresse nella notazione delle grammatiche libere dal contesto sono riportante qua accanto. Le prime tre regole costituiscono la base, infatti nella parte a destra non contengono varabili. Le ultime due regole costituiscono la parte induttiva.

Per esempio, la regola 4 dice che, se si prende una qualsiasi stringa ω della classe P (ovvero delle stringhe palindrome), anche 0ω0 si trova nella classe P.

5.1.2 Definizione delle grammatiche libere dal contesto

La descrizione grammaticale di un linguaggio consiste di quattro componenti importanti:

1) Terminali (T): Un insieme finito di simboli che formano le stringhe del linguaggio da definire. Nell'esempio delle palindrome l'insieme è {0, 1}. Chiameremo quest'alfabeto i terminali o simboli terminali.
2) Variabili (V): un insieme finito di variabili, talvolta detto anche "non terminali" oppure "categoria sintattiche". Ogni variabile rappresenta un linguaggio, ossia un insieme di stringhe. Nell'esempio precedente c'è una sola variabile, P, usata per rappresentare la classe delle stringhe palindrome sull'alfabeto {0, 1}.
3) Simbolo iniziale (S): Una variabile, detta simbolo iniziale, che rappresenta il linguaggio da definire. Le altre variabili rappresentano classi ausiliarie di stringhe, che contribuiscono a definire il linguaggio del simbolo iniziale. Nell'esempio, P, l'unica variabile, è il simbolo iniziale
4) Insieme delle produzioni (P): Un insieme finito di produzioni, o regole, che rappresentano la definizione ricorsiva di un linguaggio. Ogni produzione consiste di tre parti:
    a. Una variabile che viene definita (parzialmente) dalla produzione ed è spesso detta la testa della produzione
    b. Il simbolo di produzione →
    c. Una stringa di zero o più terminali e variabili, detta il corpo della produzione, che rappresenta un modo di formare stringhe nel linguaggio della variabile di testa. Le stringhe si formano lasciando immutati i terminali e sostituendo ogni variabile del corpo con una stringa appartenente al linguaggio della variabile stessa.

G = (V, T, P, S)

- V = variabili
- T = terminali
- P = produzioni
- S = simbolo iniziale

Questi quattro componenti formano una grammatica libera dal contesto (CFG). Rappresentiamo una CFG per mezzo dei suoi quattro componenti:

G = (V, T, P, S)

ESEMPIO

La grammatica G_{pal} per le palindrome è rappresentata da

G_{pal} = ({P}, {0, 1}, A, P)

Dove A rappresenta le produzioni elencate precedentemente.

ESEMPIO

Esaminiamo una CFG più complessa, che rappresenta le espressioni in un tipico linguaggio di programmazione.

1.	E	→	I
2.	E	→	E + E
3.	E	→	E * E
4.	E	→	(E)
5.	I	→	a
6.	I	→	b
7.	I	→	Ia
8.	I	→	Ib
9.	I	→	I0
10.	I	→	I1

Dapprima ci limitiamo agli operatori + e *. Ammettiamo che gli operandi siano identificatori, ma al posto dell'insieme completo di identificatori tipici (una lettera seguita da zero o più lettere e cifre) accettiamo solo le lettere a e b e le cifre 0 e 1. Ogni identificatore deve iniziare per a o b e può continuare con una qualunque stringa in {a, b, 0, 1}*.

In questa grammatica sono necessarie due variabili. La prima, che chiameremo E, rappresenta le espressioni. È il simbolo iniziale e rappresenta il linguaggio delle espressioni che stiamo definendo. L'altra variabile I rappresenta gli identificatori. È quindi definita formalmente da G = ({E, I}, T, P, E), dove T è l'insieme dei simboli {+, *, (,), a, b, 0, 1} e P e l'insieme di produzioni dell'elenco qua accanto.

La regola 1) è la base per le espressioni e afferma che un'espressione può essere un singolo identificatore.

Le regole dalla 2) alla 4) descrivono il caso induttivo per le espressioni. La regola 2) afferma che un'espressione può essere formata da due espressioni connesse dal segno +, la regola 3) dal segno *, la 4) che, se si prende una qualunque espressione e la si racchiude fra parentesi, il risultato è ancora un'espressione.

Le regole dalla 5) alla 10) descrivono gli identificatori. La base è data dalle regole 5) e 6) secondo le quali a e b sono identificatori. Le rimanenti regole sono il caso induttivo e affermano che se abbiamo un identificatore possiamo farlo seguire da a, b, 0 oppure 1 e il risultato è comunque un altro identificatore.

5.1.3 Derivazioni per mezzo di una grammatica

Le produzioni di una CFG si applicano per dedurre che determinate stringhe appartengono al linguaggio di una certa variabile.

La deduzione può seguire due strade:

- **Inferenza ricorsiva**: Quella più comune si serve delle regole utilizzando il corpo per passare alla testa. In altre parole, prendiamo stringhe di cui conosciamo l'appartenenza al linguaggio di ognuna delle variabili del corpo, le concateniamo nell'ordine adeguato, e deduciamo che la stringa risultante è nel linguaggio della variabile che compare in testa
- **Derivazione**: applica le produzioni dalla testa al corpo. Espandiamo il simbolo iniziale con una delle sue produzioni, espandiamo ulteriormente la stringa risultante sostituendo una delle variabili con il corpo di una delle sue produzioni.

Il processo di derivazione di stringhe per applicazione di prodizioni dalla testa al corpo richiede la definizione di un nuovo simbolo di relazione, =>. Supponiamo che G = (V, T, P, S) sia una CFG. Sia αAβ una stringa di terminali e variabili, dove A è una variabile. In altre parole, α e β sono stringhe in (V U T)*, e A è in V. sia A → γ una produzione di G. allora scriviamo αAβ => αγβ, se G risulta chiara dal contesto, scriviamo semplicemente αAβ => αγβ. Possiamo estendere la relazione => fino a farle rappresentare zero, uno o più passi di derivazione, scrivendola come =>*.

1.	E	→	I
2.	E	→	E + E
3.	E	→	E * E
4.	E	→	(E)
5.	I	→	a
6.	I	→	b
7.	I	→	Ia
8.	I	→	Ib
9.	I	→	I0
10.	I	→	I1

ESEMPIO

L'inferenza che a*(a+b00) appartiene al linguaggio della variabile E si riflette in una derivazione della stringa, a partire dalla stringa E. ecco un esempio:

E => E * E => I * E => a * E => a * (E) => a* (E + E) => a * (I + E) => a * (a + E) => a * (a + I) => a * (a + I0) => a * (a + I00) => a * (a + b00)

Quindi E =>* a * (a + b00)

5.1.4 Derivazioni a sinistra e a destra

Per ridurre il numero di scelte possibili nella derivazione di una stringa, spesso è comodo imporre che a ogni passo si sostituisca la variabile all'estrema sinistra con il corpo di una delle sue produzioni.

Tale derivazione viene detta derivazione a sinistra (leftmost derivation) e si indica tramite le relazioni $=>_{lm}$ e $=>^*_{lm}$.

5.1.5 Il linguaggio di una grammatica

Se G = (V, T, P, S) è una CFG, il linguaggio di G, denotato con L(G), è l'insieme delle stringhe terminali che hanno una derivazione dal simbolo iniziale. In altri termini

$L(G) = \{\omega \text{ in } T^* \mid S =>_G^* \omega\}$

Se un linguaggio L è il linguaggio di una CFG si dice che è L è un linguaggio libero dal contesto (context free language – CFL).

5.1.6 Forme sentenziali

Le derivazioni dal simbolo iniziale producono stringhe che hanno un ruolo speciale e che chiameremo "forme sentenziali". Ossia, se G = (V, T, P, S) è una CFG, allora qualunque stringa α in (V U T)* tale che S =>* α è una forma sentenziale.

5.2 ALBERI SINTATTICI

L'albero mostra in modo chiaro come i simboli di una stringa terminale sono raccolti in sottostringhe, ciascuna appartenente al linguaggio di una delle variabili della grammatica.

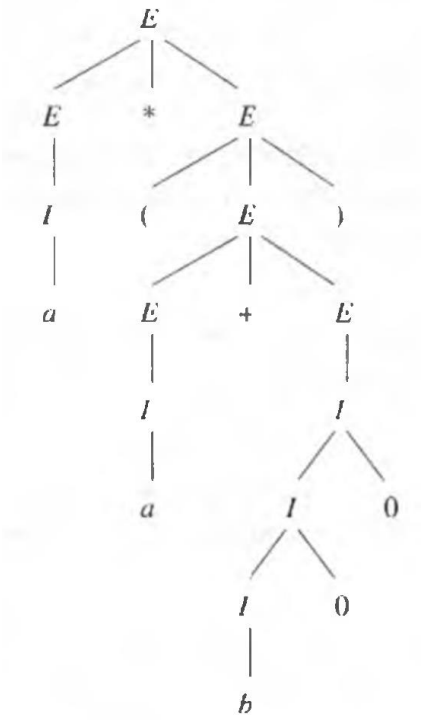
Un albero di questo tipo, detto "albero sintattico" (parse tree), sia la struttura dati ideale per rappresentare il programma sorgente in un compilatore.

5.2.1 Costruzione di alberi sintattici

Fissiamo una grammatica G = (V, T, P, S). Gli alberi sintattici di G sono alberi che soddisfano le seguenti condizioni:

1) Ciascun nodo interno è etichettato da una variabile in V
2) Ciascuna foglia è etichettata da una variabile, da un terminale o da ε. Se una foglia è etichettata ε, deve essere l'unico figlio del suo genitore
3) Se un nodo interno è etichettato A e i suoi figli sono etichettati, a partire da sinistra, X_1, X_2, …, X_k, allora A → $X_1X_2…X_k$ è una produzione in P. Si noti che un X può essere ε solo nel caso in cui è l'etichetta di un figlio unico, e quindi A → ε è una produzione di G.

5.2.2 Il prodotto di un albero sintattico



Se concateniamo le foglie di un albero sintattico a partire da sinistra otteniamo una stringa, detta il prodotto dell'albero, che è sempre una stringa derivata dalla variabile della radice. Di particolare importanza sono gli alberi sintattici che rispettano queste due condizioni:

1) Il prodotto è una stringa terminale, ovvero tutte le foglie sono etichettate da un terminale o da ε
2) La radice è etichettata dal simbolo inziale

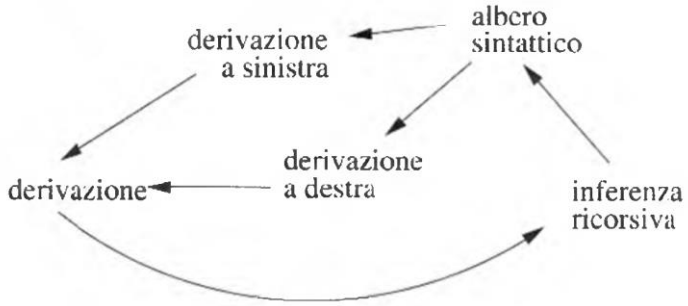
Questi sono gli alberi sintattici i cui prodotti sono stringhe nel linguaggio della grammatica associata.

Qua accanto l'albero sintattico che illustra come a * (a + b00) sia nel linguaggio della grammatica delle espressioni.

5.2.3 Inferenza, derivazioni e alberi sintattici

Data una grammatica G = (V, T, P, S), dimostriamo che i seguenti enunciati si equivalgono:

1) La procedura di inferenza ricorsiva stabilisce che la stringa terminale ω è nel linguaggio della variabile A
2) A =>* ω
3) A $=>_{lm}$* ω
4) A $=>_{rm}$* ω
5) Esiste un albero sintattico con radice A e prodotto ω.



Se escludiamo l'inferenza ricorsiva, definita solo per stringhe terminali, le altre condizioni sono equivalenti anche se ω contiene variabili.

5.2.4 Degli alberi alle derivazioni

Vediamo ora come si costruisce una derivazione a sinistra a partire da un albero sintattico.

Per capire come si costruiscono le derivazioni, consideriamo in primo luogo come la derivazione di una stringa da una variabile può essere incorporata in un'altra derivazione.

Esempio

1.	E	→	I
2.	E	→	E + E
3.	E	→	E * E
4.	E	→	(E)
5.	I	→	a
6.	I	→	b
7.	I	→	Ia
8.	I	→	Ib
9.	I	→	I0
10.	I	→	I1

Si può facilmente verificare che esiste una derivazione E => I => Ib => ab

Di conseguenza per tutte le stringhe α e β è vero anche che

αEβ => αIβ => αIaβ => αabβ

se abbiamo una derivazione che comincia con E => E + E => E + (E) possiamo applicare la derivazione di ab dalla seconda E trattando "E + (" come se fosse α e ")" come se fosse β.

Ora possiamo dimostrare un teorema che ci permette di convertire un albero sintattico in una derivazione a sinistra. La dimostrazione è un'induzione sull'altezza dell'albero, ovvero sulla sua lunghezza massima di un cammino che parte dalla radice e procede verso il basso.

5.3 Applicazioni delle grammatiche libere dal contesto

Le grammatiche si impiegano per descrivere i linguaggi di programmazione, ma un aspetto più importante è la possibilità di trasformare automaticamente una CFG in un parser.

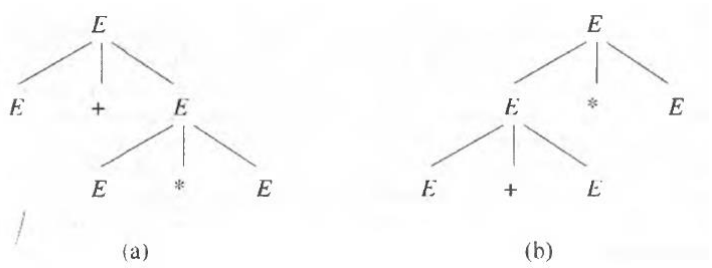
È facile prevedere lo sviluppo di un XML agevolerà il commercio elettronico permettendo ai partner di condividere convenzioni sul formato degli ordini, delle descrizioni dei prodotti e di altri tipi di documenti. Una parte essenziale di XML è la DTD, in sostanza una grammatica libera dal contesto per descrivere i tag ammessi e i modi in cui essi possono strutturarsi. I tag sono parole racchiuse fra parentesi angolari, per esempio come in HTML.

5.4 Ambiguità nelle grammatiche e nei linguaggi

Se una grammatica non genera strutture univoche, talvolta possiamo modificarla per raggiungere lo scopo. Purtroppo in certi casi l'operazione non è possibile. Ci sono infatti dei CFL inerentemente ambigui: ogni grammatica di un tale linguaggio associa più di una struttura ad alcune stringhe del linguaggio.

5.4.1 Grammatiche ambigue

Riprendendo la grammatica vista prima, questa genera espressioni con qualsiasi sequenza degli operatori * e +; le produzioni E → E+E | E*E permettono di generarle in qualsiasi ordine.



Prendiamo per esempio la forma sentenziale E + E * E ce ha due derivazioni da E:

1) E => E + E => E + E * E
2) E => E * E => E + E * E

La differenza fra le due derivazioni è rilevante. Le due derivazioni, per quanto sembrino simili, in realtà dicono:

1) E + (E * E)
2) E * (E + E)

Quindi questa grammatica non genera strutture univoche. Per poter impiegare questa grammatica in un compilatore dovremmo modificarla in modo che generi solo il raggruppamento corretto.

L'esistenza di derivazioni distinte per la stessa stringa non comporta di per sé un difetto della grammatica. L'ambiguità non risulta da derivazioni multiple, ma dall'esistenza di due o più alberi sintattici. Perciò diciamo che una CFG G = (V, T, P, S) è ambigua se esiste almeno una stringa ω in T* per la quale possiamo trovare due alberi sintattici distinti.

5.4.2 Eliminare le ambiguità da una grammatica

Non esiste un algoritmo che ci dica se una CFG sia ambigua e ci sono linguaggi che sono inerentemente ambigui.

Ci sono alcune tecniche assodate per l'eliminazione delle ambiguità.

Partendo dall'albero di prima, il problema di imporre una precedenza si risolve introducendo alcune variabili, ognuna delle quali rappresenta le espressioni con lo stesso grado di "forza di legamento", secondo lo schema seguente:

- Un fattore è un'espressione che non si può scomporre rispetto a un operatore adiacente, * o +. Nel linguaggio delle espressioni i soli fattori sono i seguenti:
    - Identificatori: non è possibile separare le lettere di un identificatore inserendo un operatore
    - Qualsiasi espressione fra parentesi, indipendentemente dal suo contenuto. Lo scopo delle parentesi è proprio quello di impedire che ciò che racchiudono diventi l'operando di un operatore esterno.
- Un termine è un'espressione che non può essere scomposta dall'operatore +. Nell'esempio, in cui + e * sono i soli operatori, un termine è il prodotto di uno o più fattori.
- Per espressione intenderemo d'ora in avanti qualsiasi espressione, comprese quelle che possono essere spezzate da un * o da un + adiacente.

5.4.3 Derivazioni a sinistra come modo per esprimere l'ambiguità

Le derivazioni non sono necessariamente uniche, anche in grammatiche non ambigue, ma in una grammatica non ambigua le derivazioni a sinistra sono uniche, così come le derivazioni a destra.

Teorema: per ogni grammatica G = (V, T, P, S) e per ogni stringa ω in T*, ω ha due alberi sintattici distinti se e solo se ha due distinte derivazioni a sinistra da S.

5.4.4 Ambiguità inerente

Un linguaggio L libero dal contesto si dice "inerentemente ambiguo" se tutte le sue grammatiche sono ambigue. Se anche una sola grammatica per L non è ambigua, L non è ambiguo.

Esempio

$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

Quindi, nei due linguaggi:

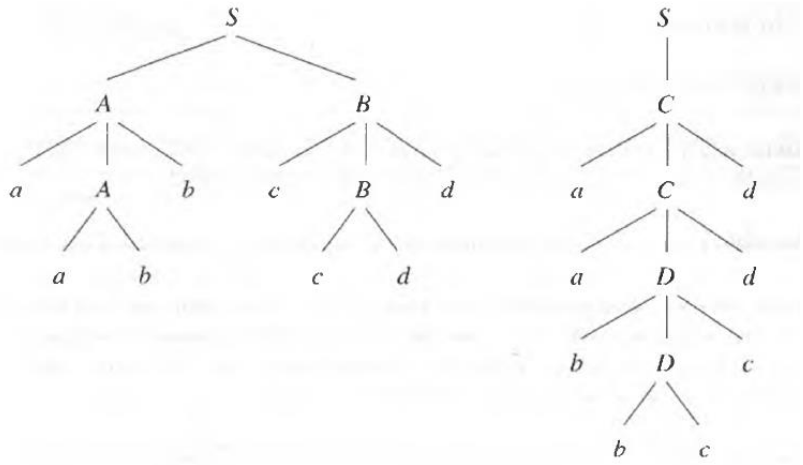
1) ci sono tanti a quanti b e tanti c quanti d
2) ci sono tanti a quanti d e tanti b quanti c

il linguaggio è libero dal contesto. La grammatica più evidente per L impiega insiemi separati di produzioni per i due tipi di stringhe.

S → AB | C
A → aAb | ab
B → cBd | cd
C → aCd | aDd
D → bDc | bc

Questa grammatica è ambigua. Per esempio, la stringa aabbccdd ha due derivazioni a sinistra:

1) $S \Rightarrow_{lm} AB \Rightarrow_{lm} aAbB \Rightarrow_{lm} aabbB \Rightarrow_{lm} aabbcBd \Rightarrow_{lm} aabbccdd$
2) $S \Rightarrow_{lm} C \Rightarrow_{lm} aCd \Rightarrow_{lm} aaDdd \Rightarrow_{lm} aabDcdd \Rightarrow_{lm} aabbccdd$



la dimostrazione del perché tutte le grammatiche di l siano ambigue è complicata. Fondamentalmente dobbiamo provare che, fatta eccezione per un numero finito di stringhe, tutte le stringhe contenenti lo stesso numero di a, b, c, d possono essere generate in due modi diversi: nel primo caso si fa in modo che il numero di a sia uguale al numero di b e il numero di c uguale al numero di d, nel secondo caso si generano tanti a quandi d e tanti b quanti c.

6 AUTOMI A PILA

I linguaggi liberi dal contesto sono deifniti da automi del tipo “a pila”, un’estensione degli automi a stati finiti non deterministici con epsilon-transizioni.

Un automa a pila è un epsilon-NFA con l’aggiunta di uno stack sul quale si può:

- Leggere
- Inserire
- Eliminare

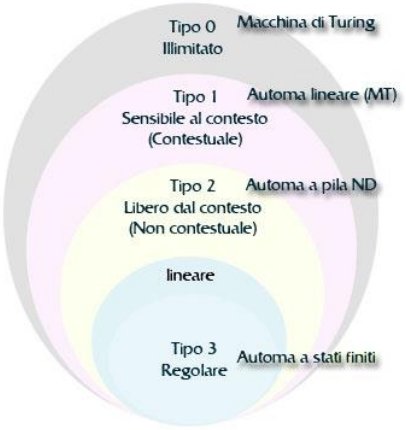
Agendo solo sulla sommità dello stack.

Esistono due tipi di automi a pila:

- NPDA: automi a pila non deterministici
- DPDA: automi a pila deterministici, sottoinsieme dei NPDA

6.1 DEFINIZIONE DI AUTOMA A PILA

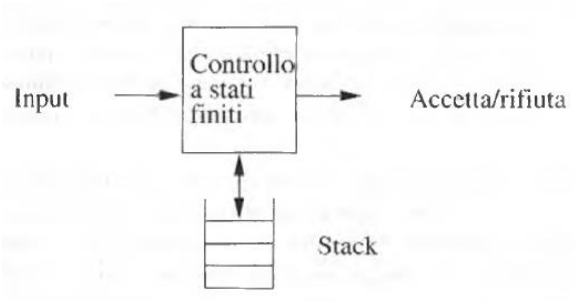
6.1.1 Introduzione formale



Un automa a pila è un automa a stati finiti non deterministico on epsilon-transizioni e uno stack aggiuntivo per memorizzare una stringa di simboli. Quindi, a differenza degli automi a stati finiti, gerazie allo stack l’automa acquisisce una struttura dati con criterio lifo.

Gli automi a pila riconoscono tutti e solo i linguaggi liberi dal contesto (tipo 2).

Possiamo immaginare un automa a pila come nella figura qua accanto. L’automa a pila osserva il simbolo alla sommità dello stack e può basare la transizione sullo stato corrente, sul simbolo di input e sul simbolo alla sommità dello stack.



un esempio di linguaggio non libero dal contesto è $\{0^n1^n2^n \mid n \geq 1\}$, ovvero l’insieme delle stringhe che contengono gruppi uguali di 0, 1 e 2.

In alternativa alle transizioni descritte prima, l’automa può anche avere una transizione “spontanea” usando epsilon come input in luogo di un simbolo effettivo. In una transizione l’automa compie tre operazioni:

- Consuma dall’input il simbolo che usa nella transizione
- Passa a un nuovo stato, che può essere uguale o no al precedente
- Sostituisce il simbolo in cima allo stack con una stringa. La stringa può essere epsilon, che corrisponde a togliere dallo stack, lo stesso simbolo che c’era sulla sommità o un altro simbolo con l’effetto di trasformare la sommità dello stack.

6.1.2 Definizione formale di automa a pila

La notazione formale per un automa a pila include sette componenti:

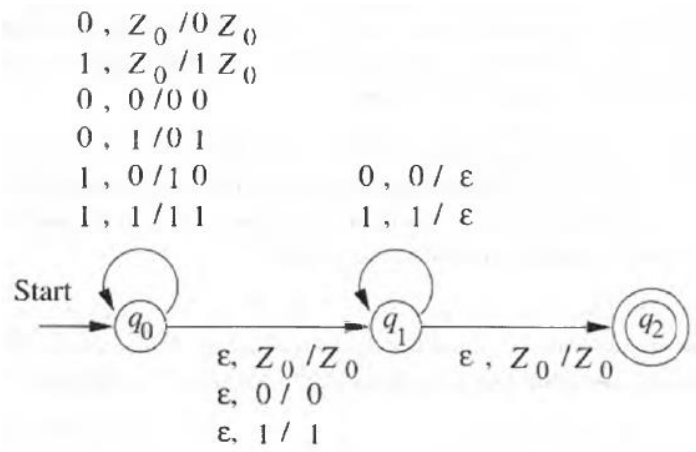
$$P = (Q, \sum, \Gamma, \delta, q_0, Z_0, F)$$

Dove:

- Q: un insieme finito di stati
- ∑: un insieme finito di simboli di input
- Γ: un alfabeto di stack, finito. Questo elemento non ha analoghi negli automi a stati finiti.
- δ: insieme delle funzioni di transizioni
- q_0: lo stato iniziale
- Z_0: simbolo iniziale di stack
- F: insieme degli stati accettanti

6.1.3 Una notazione grafica per i PDA

- I nodi corrispondono agli stati del PDA
- Una freccia etichettata start inidica lo stato inziiale mentre gli stati accettanti sono contrassegnati da un doppio cerchio
- Gli archi corrispondono alle transizioni del PDA. Un arco etichettato "a, X/b" vuol dire che leggendo l'input a e avendo X sulla sommità dello stack, la transizione posizionerà b in cima allo stack e passerà di stato



L'unica informazione che il diagramma non fornisce è il simbolo iniziale dello stack. Per convenzione usiamo Z_0.

6.1.4 Descrizioni istantanee di un PDA

La configurazione di un PDA comprende sia lo stato sia il contenuto dello stack. Di conseguenza rappresentiamo una configurazione come una tripla (q, ω, γ), dove:

- q è lo stato
- ω è l'input residuo
- γ è il contenuto dello stack

per convenzione mostriamo la sommità dello stack all'esremo sinistro di γ e il fondo all'estremo destro. La tripla è detta "descrizione istantanea" (ID) dell'automa a pila.

Nello studio dei PDA hanno particolare importanza tre principi relativi alle ID e alle transizioni:

- se una sequenza di ID è lecita per un PDA P, allora è lecita anche la computazione formata accodando una stringa all'input in ogni ID

- se una computazione è lecita per un PDA P, allora è lecita anche la computazione formata aggiungendo gli stessi simboli sotto quelli nello stack in ogni ID
- se una computazione è lecita per una PDA P, e resta una coda di input non consumata, possiamo rimuovere il residuo dall'input in ogni ID e ottenere una computazione lecita

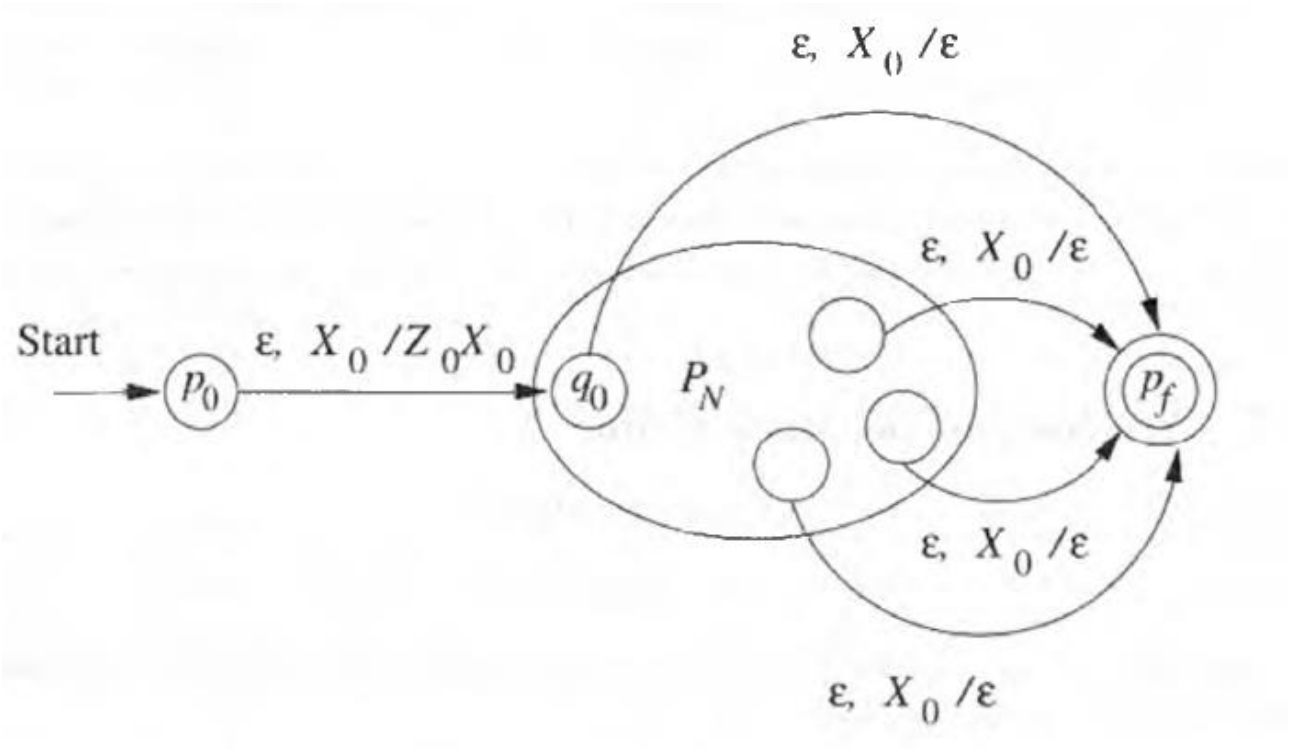
6.2 I LINGUAGGI DI UN PDA

Abbiamo stabilito che un PDA accetta uns tringa consumandola ed entrando in uno stato accettante. Chiamiamo questa soluzione "accettazione per stato finale".

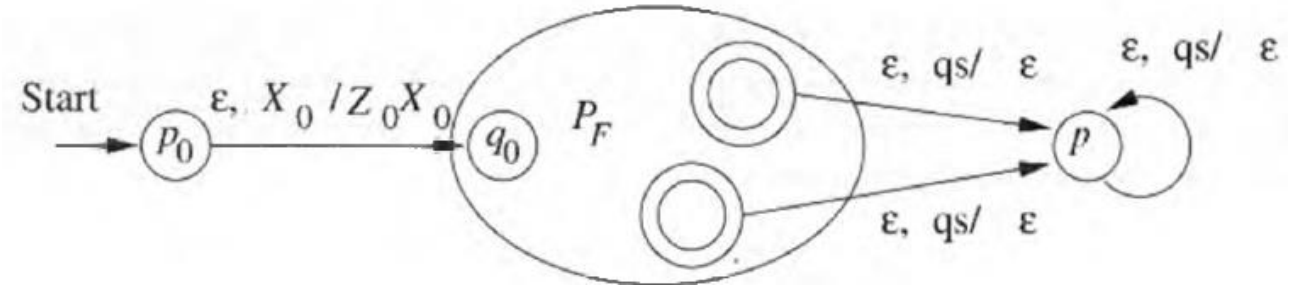
Per un PDA possiamo definire il linguaggio "accettato per stack vuoto", cioè l'insieme delle stringhe che portano il PDA a vuotare lo stack.

I due metodi sono equivalenti e possono essere convertiti tra di loro tranne in un caso: se il linguaggio non è prefix-free possiamo usare solo PDA per stato finale in quanto non sapremmo quando svuotare lo stack nel caso.

Da stack vuoto a stato finale:



Da stato finale a stack vuoto:

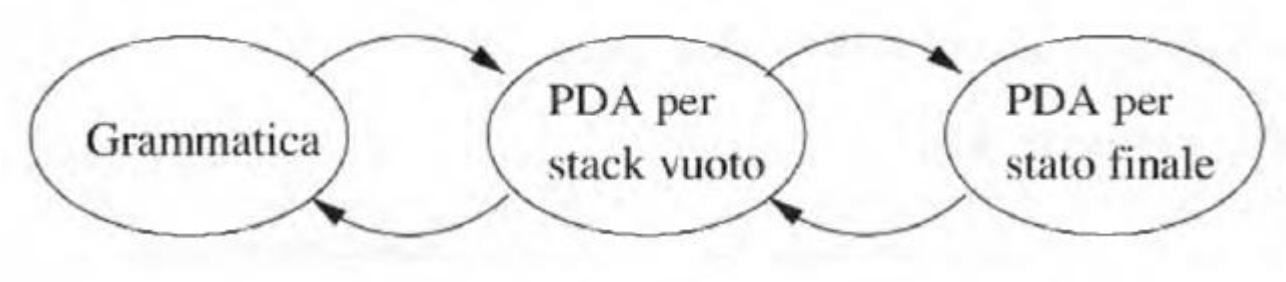


6.3 EQUIVALENZA DI PDA E CFG

Lo scopo è quello di provare che le tre classi di linguaggi che seguono coincidono:

1) I linguaggi liberi dal contesto, cioè quelli definiti dalle CFG
2) I linguaggi accettati per stato finale da un PDA
3) I linguaggi accettati per stack vuoto da un PDA

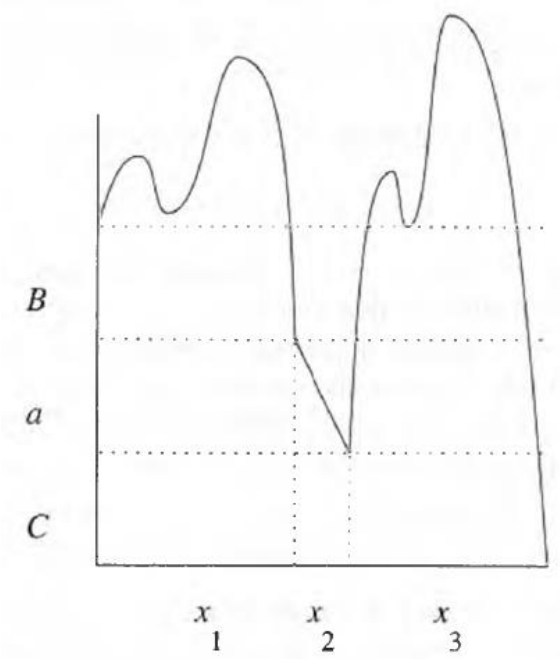
Sappiamo che 2 e 3 coincidono, quindi se riusciamo a dimostrare l'equivalenza tra 1 e 3 possiamo dare l'equivalenza tra tutti.

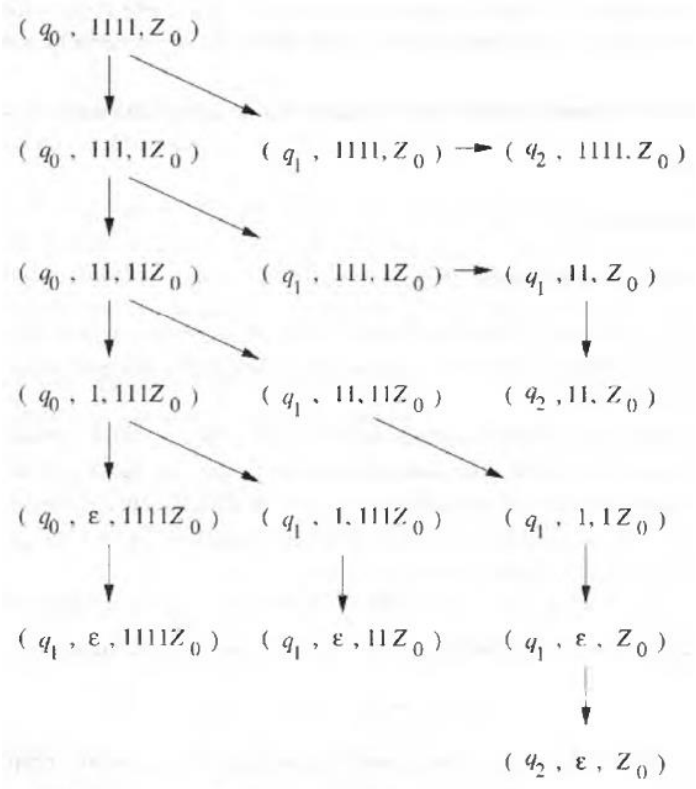


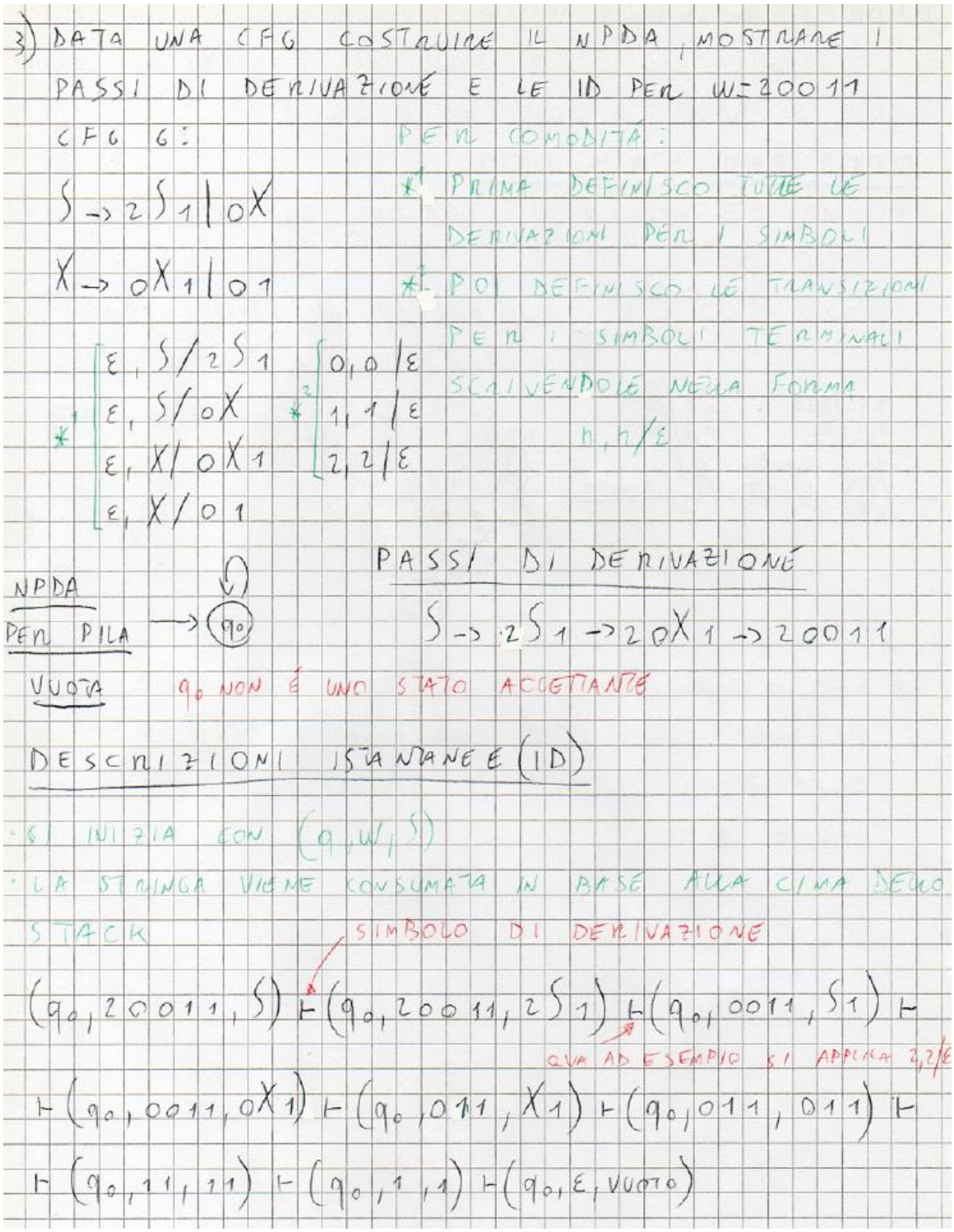
6.3.1 Dalle grammatiche agli automi a pila

Per spiegare la costruzione utilizzo un esempio:

date le produzioni di una grammatica, possiamo consumare tramite uno stack la stringa. L'automa può controllare, grazie alle transizioni, in quale stato può trovarsi e consumare di conseguenza l'input. Tramite l'albero di produzione possiamo vedere ogni singolo caso e solo quelli che arrivano in uno stato accettante sono corretti. Nell'esempio abbiamo una grammatica con produzioni a sinistra.







6.3.2 Dai PDA alle grammatiche

Nell'eliminare un simbolo di stack, un PDA può cambiare stato, dobbiamo quindi tenere traccia dello stato in cui entra quando scende di un livello nello stack.

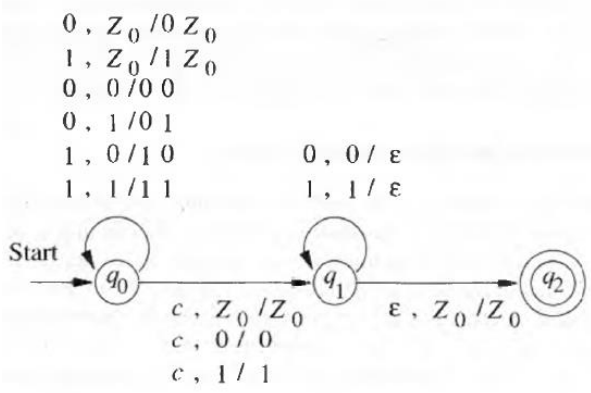
La costruzione di una grammatica equivalente impiega variabili che rappresentano ciascuna un "evento" con due componenti:

- L'eliminazione effettiva di un simbolo X dallo stack
- Il passaggio dallo stato p allo stato q, dopo la sostituzione di X con epsilon sullo stack.

6.4 AUTOMI A PILA DETERMINISTICI

I PDA possono essere non deterministici, il caso di automa deterministico è importante, soprattutto in quanto i parser si comportano generalmente come PDA deterministici. La classe dei linguaggi accettati da questi automi è quindi interessante perché ci aiuta a capire quali costrutti sono adatti ai linguaggi di programmazione.

6.4.1 Definizione di PDA deterministico



molto semplicemente un PDA è deterministico (DPDA) se per ogni coppia di input e stack vi è al massimo una sola mossa possibile.

6.4.2 Linguaggi regolari e PDA deterministici

I DPDA accettano una classe di linguaggi che si pone tra i linguaggi regolari e i CFL.

Un linguaggio L ha la proprietà di prefisso se in esso non esistono due stringhe x e y diverse, tali che x è un prefisso di y.

Teorema: un linguaggio L è N(P) per un DPDA P se e solo se L gode della proprietà di prefisso ed L è L(P') per un DPDA P'.

Quindi i DPDA accettano (per stato finale) tutti i linguaggi regolari e alcuni linguaggi non regolari. I linguaggi dei DPDA sono liberi dal contesto e hanno tutti CFG non ambigue. Perciò si collocano tra i linguaggi regolari e i linguaggi liberi dal contesto.

7 MACCHINE DI TURING: INTRODUZIONE

7.1 PROBLEMI CHE I CALCOLATORI NON POSSONO RISOLVERE

Possiamo pensare che simulando un programma si possa stabilire che cosa fa, ma dobbiamo fare i conti con programmi che impiegano un tempo straordinariamente lungo per produrre un output. Questo problema è la ragione principale della nostra incapacità di stabilire che cosa fa un programma.

Dimostrare formalmente che non esiste alcun programma in grado di svolgere un certo compito è difficile.

7.1.1 Programmi che stampano "ciao, mondo"

Il fatto che un programma stampi "Ciao, mondo" non è ovvio. Ad esempio un programma che prende input n e cerca soluzioni intere positive all'equazione x^n + y^n = z^n[6]. Se ne trova una stampa "Ciao, mondo", altrimenti la ricerca non ha mai fine e il programma non arriva a stampare la frase. Per un input n>2 il programma non troverà mai una tripla di interi positivi che soddisfi l'equazione, quindi non avrà mai termine.

7.1.2 Un ipotetico verificatore di ciao-mondo

Figura 8.3 Un ipotetico programma H che funge da verificatore di ciao-mondo.

L'impossibilità di compiere la verifica si dimostra per assurdo. In altre parole, supponiamo esista un programma H che prende come input un programma P e un input I e dice che se I è in L(P), allora stampa "Ciao, mondo".

Se un problema con algoritmo H ha sempre output certo "sì" o "no" è detto decidibile, negli altri casi è indecidibile.

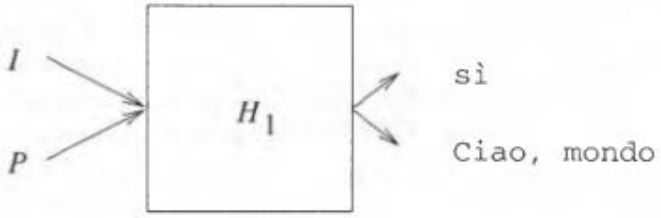


Figura 8.4 H_1 si comporta come H, ma stampa `Ciao, mondo` anziché `no`.

Il nostro obiettivo è dimostrare che H non esiste, ovvero il problema è indecidibile.

Assumiamo ora che H esista e modifichiamo l'output da "no" a "Ciao, mondo" e chiamiamo il nuovo algoritmo H1.

6 Questo è l'ultimo teorema di Fermat e sostiene che non esistono soluzioni intere all'equazione per n > 2

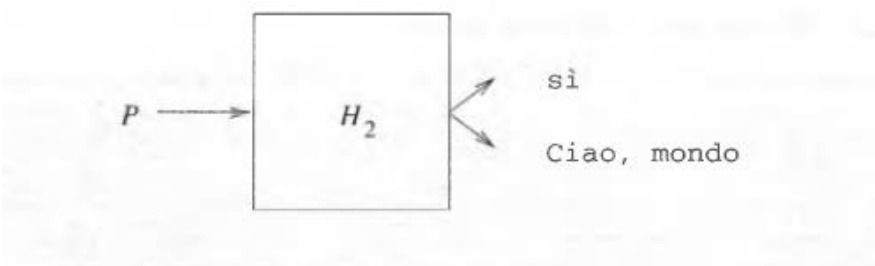


Figura 8.5 H_2 si comporta come H_1, ma usa il suo input P sia per P sia per I.

Ora riduciamo ancora di più l'input della macchina togliendo I e il nuovo programma determin che cosa farebbe P se l'input fosse il suo stesso codice, cioè che cosa farebbe H1 avendo in input P sia come programma sia come I.

quindi H2:

- copia P in un array e lo usa come input
- simula H1 e usa la copia nell'array al posto di I

a questo punto possiamo dimostrare che H2 non può esistere. Di conseguenza H1 non esiste così come H.

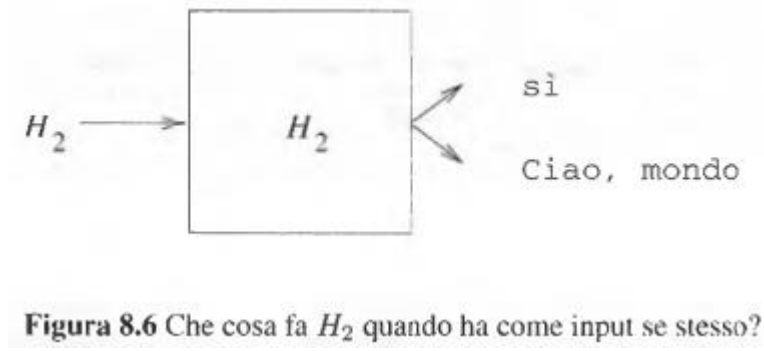


Figura 8.6 Che cosa fa H_2 quando ha come input se stesso?

la base del ragionamento sta nel capire cosa fa H2 quando gli si dà se stesso come input.

Ricordiamo ora che H2, dato un qualunque programma P:

- produce sì se P stampa "Ciao, mondo" quando gli viene dato sé stesso come input
- produce "ciao, mondo" se P, dato sé stesso, NON stampa "Ciao, mondo"

quindi ci troviamo in una situazione paradossale: supponiamo che H2 sopra produca sì, quindi H2 quando riceve sé stesso stampa Ciao, mondo, eppure abbiamo appena detto che H2 ha stampato sì. Possimo concludere che H2 non esiste.

7.1.3 Ridurre un problema a un altro

un problema che non può essere risolto da un computer è detto indecidibile. Supponiamo di voler determinare se un qualche altro problema sia o no risolvibile. Possiamo risolvere questo problema con una tecnica simile a quella vista prima, ovvero assumiamo che esista un programma in grado di risolverlo e sviluppiamo un programma paradossale che deve fare due cose contraddittorie. Basta mostrare che se potessimo risolvere il nuovo problema, allora potremmo usare tale soluzione per risolvere il problema che già sappiamo essere indecidibile.

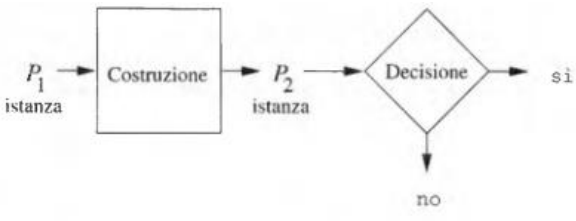


Figura 8.7 Se siamo in grado di risolvere il problema P_2, possiamo usare la soluzione per risolvere il problema P_1.

Questa tecnica è detta "riduzione di P1 a P2".

Supponiamo di sapere che P1 è indecidibile e sia P2 un nuovo problema di cui vogliamo dimostrare l'indecidibilità.

Il programma "decisione" stampa "sì" o "no" a seconda che l'istanza del problema P2 che gli fa da input si trovi o no nel linguaggio del programma stesso.

Per aiutarci usiamo un blocco chiamato "costruzione" che trasforma le stringhe di P1 in stringhe di P2. Una stringa non appartenente a L(P1) viene trasformata in una stringa non appartenente a L(P2).

Quindi per dire che la stringa di P2 è nel suo linguaggio dovremmo sapere che la stringa di P1 è nel suo linguaggio, problema che abbiamo definito per ipotesi indecidibile. Quindi, di conseguenza, anche P2 è indecidibile.

Questo è spiegato direttamente dal teorema di Rience che vedremo più avanti.

7.2 LA MACCHINA DI TURING

Una macchina di Turing è, in sostanza, un automa a stati finiti con un nastro di lunghezza infinita su cui leggere o scrivere dati.

7.2.1 Notazione per la macchina di Turing

La macchina consiste di un controllo finito, che può trovarsi in uno stato, scelto in un insieme finito. C'è un nastro diviso in caselle, o celle; ogni cella può contenere un simbolo scelto in un insieme finito.

L'input, una stringa composta dall'alfabeto di input, viene posto sul nastro, mentre tutte le altre caselle presentano un simbolo speciale, ovvero quello di blank, simbolo di nastro ma non di input.

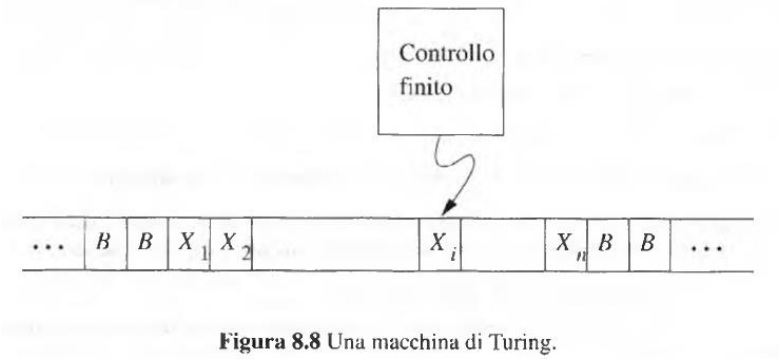


Figura 8.8 Una macchina di Turing.

in una mossa la macchina di Turing compie tre azioni:

- Cambia lo stato (può coincidere con il precedente)
- Scrive un simbolo di nastro nella cella che guarda, sostituendo quello che era presente precedentemente (può coincidere con il precedente)
- Muove la testina verso sinistra oppure verso destra

Formalmente, una macchina di Turing (MT) è:

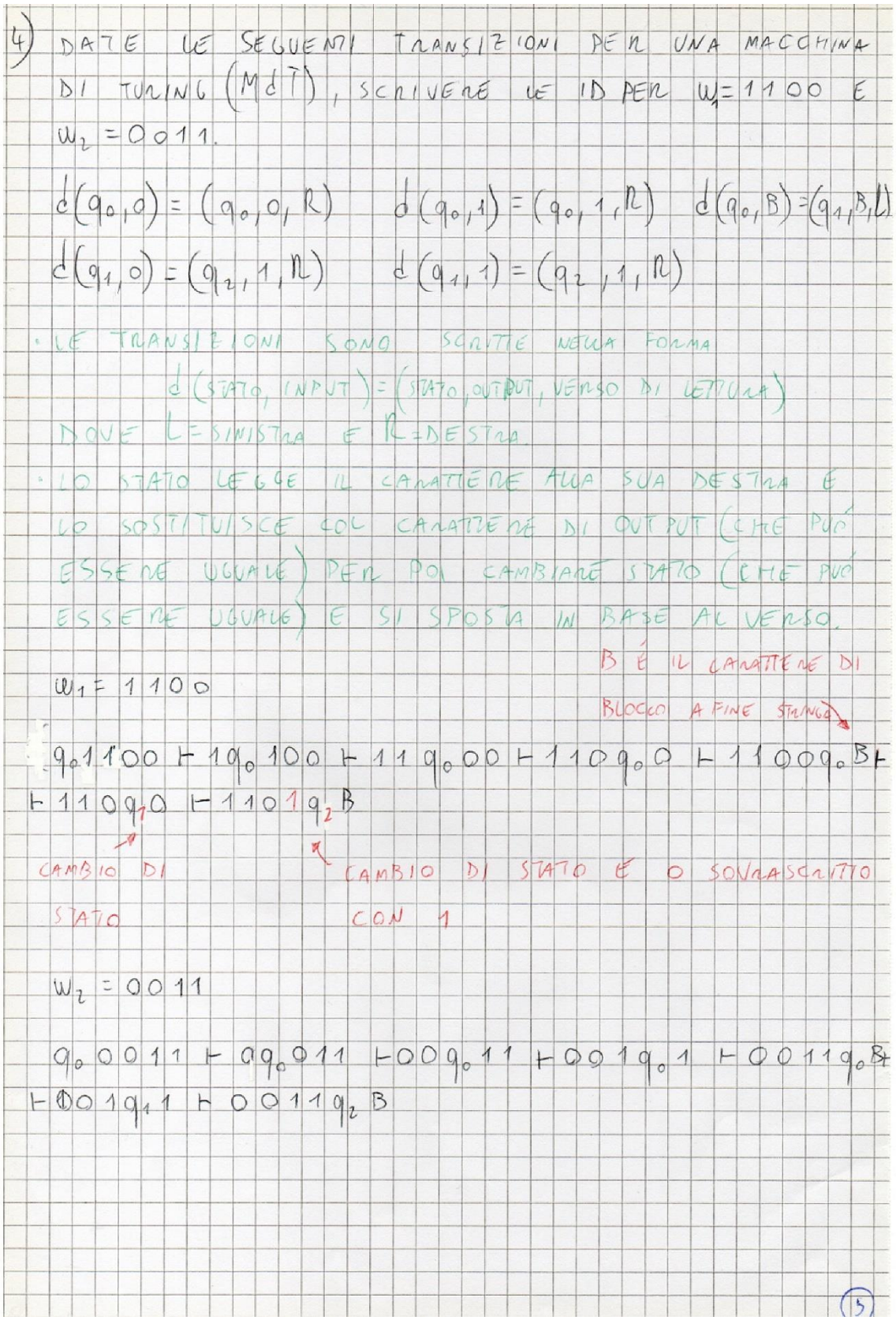
M = (Q, ∑, Γ, δ, q0, B, F)

Dove:

- Q: insieme degli stati del controllo
- ∑: insieme dei simboli di input
- Γ: insieme dei simboli di nastro, ∑ è sottoinsieme di Γ
- δ: insieme delle funzioni di transizione:
  - una transizione della MT è nella forma δ(q_iniziale, input) = (q_nuovo, carattere_nuovo, direzione)
- q0: stato inziale
- B: simbolo di blank, presente in Γ ma non in ∑
- F: insieme degli stati finali o accettanti

7.2.2 Descrizioni istantanee delle macchine di Turing

Per descrivere formalmente che cosa fa una macchina di Turing dobbiamo sviluppare una notazione per le descrizioni istantanee (ID).



7.2.3 Il linguaggio di una macchina di Turing

I linguaggi che possiamo accettare usando una macchina di Turing sono spesso denominati “linguaggi ricorsivamente enumerabili” o linguaggi RE.

7.2.4 Le macchine di Turing e l'arresto

Per le macchine di Turing si usa comunemente un'altra notazione di "accettazione": l'accettazione per arresto. Diremo che una MT si arresta se entra in uno stato q guardando un simbolo di nastro X e non ci sono mosse in questa situazione.

Il problema è che non sempre le macchine di Turing si arrestano. Se c'è sempre la certezza di arresto allora il linguaggio della MT è un linguaggio "ricorsivo", se si arresta solo in caso di accettazione è "RE ma non ricorsivo", altrimenti è indecidibile.

7.3 ESTENSIONI ALLA MACCHINA DI TURING SEMPLICE

7.3.1 Macchine di Turing multinastro

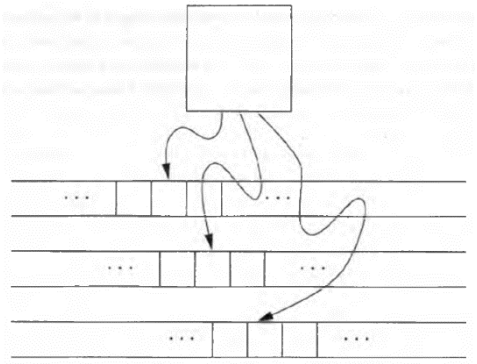


Figura 8.16 Una macchina di Turing multinastro.

una MT multinastro ha un controllo finito e un numero finito di nastri che si comportano come quelli di una MT mononastro.

La differenza consiste che una mossa di una MT multinastro dipende dallo stato e dai simboli letti da ciascuna testina e quindi:

- Cambia lo stato (può coincidere con il precedente) **su ogni nastro**
- Scrive un simbolo di nastro nella cella che guarda, sostituendo quello che era presente precedentemente (può coincidere con il precedente) **su ogni nastro**
- Muove la testina verso sinistra oppure verso destra **su ogni nastro, oppure rimane ferma (simbolo S)**

7.3.2 Equivalenza di macchine di Turing mononastro e multinastro

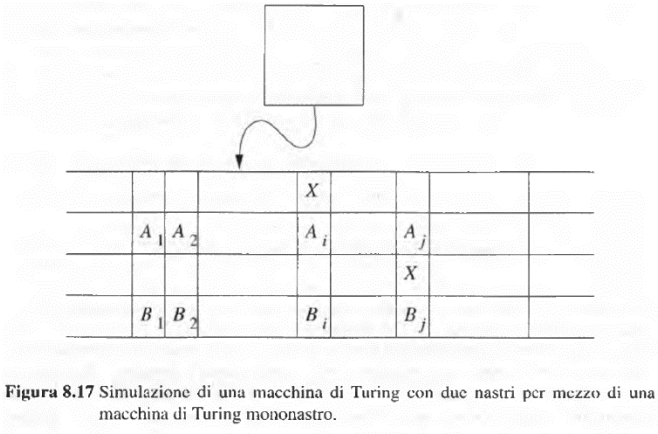


Figura 8.17 Simulazione di una macchina di Turing con due nastri per mezzo di una macchina di Turing mononastro.

ricordiamo che i linguaggi RE sono quelli accettati da una MT mononastro. Ovviamente le MT multinastro accettano tutti i linguaggi RE, perché una MT mononastro è un caso particolare di multinastro.

Possiamo però chiederci: esistono linguaggi non RE ma accettati da una MT multinastro? No.

Teorema: ogni linguaggio accettato da una MT multinastro è RE.

La dimostrazione consiste nel convertire i k nastri in 2k tracce su di un solo nastro. Metà delle tracce replica i nastri di M, le altre indicano la posizione corrente della testina sul corrispondente nastro di M.

7.3.3 Macchine di Turing non deterministiche

Una MT non deterministica (NMT) si distingue da quella deterministica nella funzione di transizione che può vedere un numero k>0 output per un solo input.

Le NMT accettano linguaggi che possono essere accettati da MT deterministiche.

Teorema: se Mn è una NMT, esite una MT deterministica Md tale L(Mn) = L(Md).

La dimostrazione consiste nel costruire Md come una MT multinastro dove il rpimo nastro contiene una sequenza di ID di Mn che comprendono lo stato di Mn. Una ID di Mn è contrassegnata come "corrente".

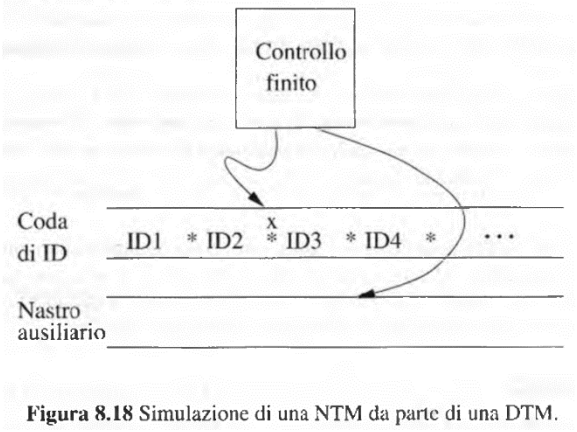


Figura 8.18 Simulazione di una NTM da parte di una DTM.

Per elaborare la ID corrente, Md compie quattro azioni:

1) Esamina lo stato e il simbolo guardato dalla ID corrente. Nel controllo di Md sono incorporate le scelte di mossa di Mn per ogni stato e simbolo
2) Se lo stato non è accettante e la combinazione stato-simbolo permette k mosse, Md copia la ID sul secondo nastro e fa k copie della ID in coda alla sequenza sul primo
3) Md modifica ognuna delle k ID secondo una delle k scelte di mossa
4) Md torna alla ID corrente contrassegnata in precedenza, cancella il contrassegno e lo sposta alla successiva ID a destra.

7.4 MACCHINE DI TURING RIDOTTE

7.4.1 Macchine di Turing con nastri semi-infiniti

In questo caso supponiamo che il nastro abbia inizio (prima casella puntata dal controllo finito) ma non fine.

X_0	X_1	X_2	...
*	X_{-1}	X_{-2}	...

È basato su due nastri:

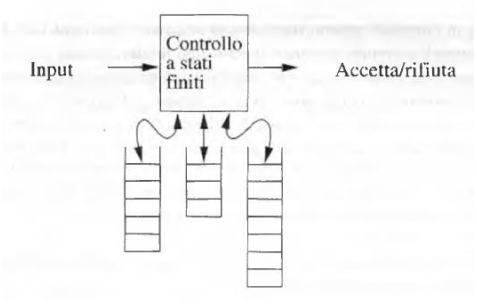
- La traccia superiore rappresenta la cella su cui si trova la testina e quelle alla sua destra
- La traccia inferiore rappresenta le celle a sinistra della posizione iniziale, in ordine inverso
- Il simbolo * segnala la fine della traccia

Imponiamo anche il divieto di scrivere blank.

Teorema: ogni linguaggio accettato da una MT M2 è accettato anche da una MT a nastri semi-infiniti con i seguenti vincoli:

- La testina di M1 non va mai a sinistra della posizione iniziale
- M1 non scrive mai un blank

7.4.2 Macchine multi-stack



una macchina multistack ha la differenza dal fatto che ogni mossa dipende da tre elementi:

- Lo stato di controllo
- L'input
- I simboli in cima a tutti gli stack

Teorema: se un linguaggio L è accettato da una MT, allora L è accettato da una macchina con due stack.

7.4.3 Macchine a contatori

Ci sono due modi per vederle:

1) Una macchina con la stessa struttura delle multistack ma con un contatore al posto di ogni stack, quindi la mossa dipende da quali contatori hanno valore nullo, compiendo due azioni:

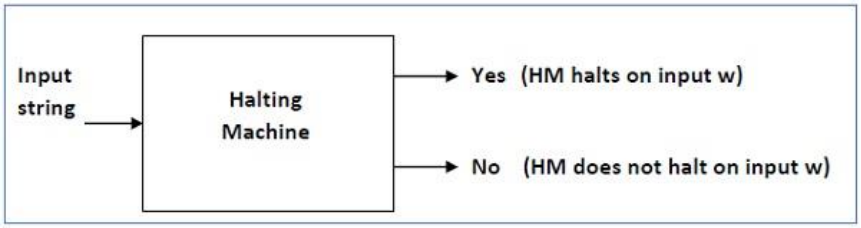
   a. Cambia stato
   b. Somma o sottrae 1 da un contatore. Poiché i contatori sono a interi positivi, la macchina non può sottrarre 1 da un contatore nullo
2) Una macchina multistack con i seguenti vincoli:
   a. Due soli simboli di stack, Z_0 (simbolo alla base dello stack) e X
   b. All'inizio ogni stack contiene Z_0
   c. Possiamo sostituire Z_0 con una stringa X^iZ_0 per i≥0
   d. Possiamo sostituire X solo con X^i per un i≥0

Ogni linguaggio accettato da una macchina a contatori è RE, ed è un CFL.

7.5 HALTING PROBLEM

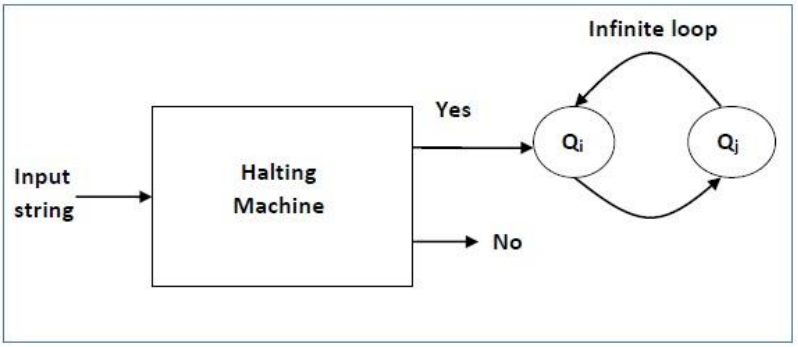
È il problema derivante dal determinare, dalla descrizione arbitraria di un programma e un input, se il programma avrà termine (halt) o no.

Prima creiamo una macchina che restituisce Sì se la macchina si ferma, no se non si ferma. Questa macchina che chiamiamo halting machine (HM) si ferma sempre e dà sempre la risposta corretta.



Ora creiamo una macchina che nega il risultato dell'HM:

- Se si ferma allora non si ferma
- Se non si ferma allora si ferma



Quindi, una HM2 che riceve in input sé stessa:

- Se si dovesse fermare in base all'input, allora non si ferma
- Se non si dovesse fermare, allora si ferma

Questa macchina è impossibile in base a questa contraddizione, quindi il programma è indecidibile.

8 INDECIDIBILITÀ

Esistono problemi non risolvibili da un computer. I limiti fisici come la dimensione dello spazio di indirizzamento non ci interessano al momento. Usando una macchina di Turing, quindi con nastro infinito, possiamo comprendere meglio il problema die problemi indecidibili.

Ad esempio, una macchina di Turing MT accetta il codice di sé stessa come input?

Distinguiamo i problemi che possono essere risolti da una macchina di Turing in due classi:

- Decidibili (ricorsivi): Quelli per cui c'è un algoritmo, ovvero una macchina di Turing si arresta a prescindere dal fatto di accettare o no il suo input
- Indecidibili: Quelli che vengono risolti solo da macchine di Turing che possono girare per sempre su input non accettati.

8.1 UN LINGUAGGIO NON RICORSIVAMENTE ENUMERABILE

Ricordiamo che un linguaggio L è ricorsivamente enumerabile (RE) se L = L(M) per una macchina di Turing M.

Il nostro obiettivo è dimostrare l'indecidibilità del linguaggio formato dalle coppie (M, ω) tali che:

1) M è una macchina di Turing (adeguatamente ocdificata in binario) con alfabeto di input {0, 1}
2) ω è una stringa di 0 e di 1
3) M accetta l'input ω

Possiamo trattare qualunque stringa binaria come se fosse una macchina di Turing. Se la stirnga non è una rappresentazione ben formata di una TM, possiamo pensarla come una TM priva di mosse.

L'obiettivo intermedio per dimostrare l'esistenza di linguaggi non RE è capire il linguaggio di diagonalizzazione (L_d), che consiste di tutte le stringhe ω tali che TM rappresentata da ω non accetta l'input ω. Dimostreremo che non c'è una TM che accetti L_d.

Ricoridamo che la non esistenza di una TM per un linguaggio è una proprietà più forte della non decidibilità, infatti in quel caso esiste una TM che non si ferma in caso di input errato.

La funzione del linguaggio L_d è come quella di un programma H1 che stampa "ciao, mondo" ogni volta che il suo input NON stampa "ciao, mondo" quando riceve sé stesso come input. Allo stesso modo L_d non può essere accettato da una macchina di Turing. Infatti, se lo fosse, la TM in questione entrerebbe in disaccordo con sé stessa nel momento in cui dovesse ricevere come input il proprio codice.

8.1.1 Enumerazione delle stringhe binarie

Nel seguito dovremo assegnare numeri interi a tutte le stringhe binarie in modo che ciascuna stringa corrisponda a un unico intero e ciascun intero corrisponda a un'unica stringa.

Le stringhe verranno elencate tutte in ordine lessicografico, iniziando dalla stringa vuota epsilon, seguita da 0, 1, 00, 01, 11, ecc.

D'ora in poi ci riferiremo alla stringa i-esima come stringa ω_i.

8.1.2 Codici per le macchine di Turing

Spiegazione più comoda:

- Ad ogni stato attribuiamo una stringa composta da 0, 0^i
- Ad ogni elemento dell'alfabeto attribuiremo una stringa composta da 0, 0^i

- Ad ogni simbolo di spostamento (R, L) attribuiremo una stringa composta da 0, 0^i
- Per dividere gli elementi di una transizione usiamo un singolo 1
- Per dividere le transizioni useremo 11
- Per chiudere la definizione della macchina di Turing useremo 111

Esempio di una macchina di Turing:

q_1	0
q_2	00
q_3	000
0	0
1	00
B	000
R	0
L	00

Transizioni	Traduzioni binarie
δ(q1, 1) = (q3, 0, R)	0100100010 10
δ(q3, 0) = (q1, 1, R)	000 10 10 10 0010
δ(q3, 1) = (q2, 0, R)	000 100 100 10 10
δ(q3, B) = (q3, 1, L)	000 1000 1000 100 100

Traduzione macchina:

0100100010101100010101001011000100100101011000100010001001001**11**1

8.1.3 Il linguaggio di diagonalizzazione

Ora che abbiamo dato una definizione binaria di TM possiamo dare una nozione concreta di M_i: la TM m il cui codice è ω_i.

Molti interi non corrispondono a nessuna TM, ad esempio 11010 non corrisponde ad alcuna macchina in quanto non comincia con 0, oppure 001011101000110 perché contiene 111.

A questo punto possiamo dare una definizione fondamentale: il linguaggio L_d, detto linguaggio di diagonalizzazione (o diagonale), è l'insieme delle stringhe ω_i tali che ω_i non è in $L(M_i)$.

In altre parole: L_d consiste in tutte le stinghe ω tali che la TM m con codice ω non accetta quando riceve ω come input.

	Macchine di Turing M_i					
Stringhe ω_j	j/i	1	2	3	4	…
	1	0	1	1	0	…
	2	1	1	0	0	…
	3	0	0	1	1	…
	4	0	1	0	1	…
	…	…	…	…	…	…

Perché linguaggio di diagonalizzazione? Per questo motivo: vediamo la tabella qua accanto. La tabella indica per ogni i e j se la TM mi accetta la stringa di input ω_j. 1 significa "sì, lo accetta" e 0 "no, non accetta". Attenzione la tabella rappresentata non è realistica, infatti non inizia con la stringa vuota, è solo un modo per mostrare come funziona.

Quindi, i valori sulla diagonale segnalano se Mi accetta ω_j. Per costruire Ld complementiamo la diagonale, ovvero 0 diventa 1 e viceversa.

La diagonale dell'esempio è 0111, allora diventa 1000. Perciò L_d contiene ω_1 e non ω_2, ω_3, ω_4.

Il metodo di complementare la diagonale si chiama, appunto, "diagonalizzazione". Questo vettore caratteristico si trova in disaccordo con ogni riga della tabella in almeno una colonna. Quindi il complemento della diagonale non può essere il vettore caratteristico di una macchina di Turing.

8.1.4 Dimostrazione che L_d non è ricorsivamente enumerabile

Sfruttando il ragionamento riguardante i vettori caratteristici e la diagonale, dimostreremo ora che nessuna macchina di Turing accetta L_d.

Teorema: L_d non è un linguaggio ricorsivamente enumerabile. In altre parole, non esiste alcuna macchina di Turing che accetta L_d.

Dimostrazione: supponiamo che per una TM M, Ld = L(M), ovvero la macchina M accetta in input Ld. Poiché Ld è un linguaggio binario, M rientra nell'elenco di macchine di Turing che abbiamo costruito, dato che questo elenco include tutte le TM con alfabeto di input binario. Di conseguenza esiste almeno un codice per M, poniamo i, ossia M = Mi.

Chiediamoci ora se ω_i è in Ld (ovvero se ω_i non è accettata dalla MT che codifica).

- Se ω_i è in Ld, allora Mi accetta ω_i. pertanto per la definizione di Ld, ω_i non è in Ld, poiché Ld contiene solo le stringhe ω_j tali che Mi NON accetta ω_j.
- Analogamente, se ω_i NON è in Ld, allora Mi non accetta ω_i. Di conseguenza, per la definizione di Ld, ω_i è in Ld.

Poiché ω_i non può essere né non essere in Ld, abbiamo una contraddizione che M esista. In altre parole, Ld non è un linguaggio ricorsivamente enumerabile.

	Macchine di Turing M_i					
Stringhe ω_j	j/i	1	2	3	4	…
	1	acc	acc	rej	rej	…
	2	rej	acc	rej	acc	…
	3	acc	rej	acc	acc	…
	4	rej	acc	rej	rej	…
	…	…	…	…	…	…

Per spiegarla diversamente: La stringa $\omega_i \in$ Ld se e solo se Mi accetta ω_i. Ora consideriamo il complemento L = $\bar{}$Ld. Questo linguaggio non può essere riconosciuto da alcuna macchina MT. Infatti, se una macchina Ml riconoscesse L, allora consideriamo ω_k. Ci sono due possibilità:

- Se M_k accetta ω_k, allora la cella (k, k) è acc. Ciò implica che $\omega_k \notin$ Ld, ma allora M_k (che riconosce L) non deve accettare ω_k
- Se M_k non accetta ω_k, allora la cella (k, k) è reg. il che implica che $\omega_k \in$ L, ma allora M_k (che riconosce L) deve accettare ω_k

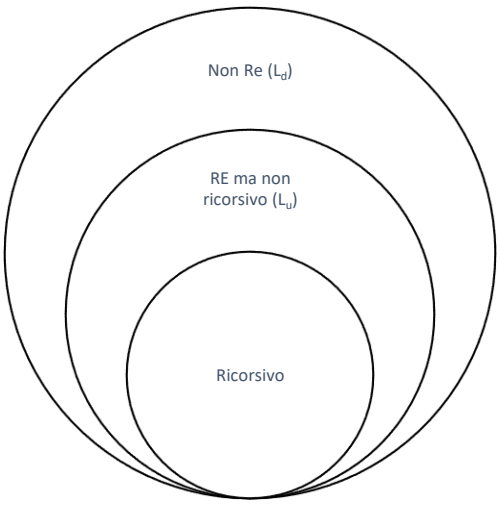
8.2 UN PROBLEMA INDECIDIBILE MA RICORSIVAMENTE ENUMERABILE

Precisiamo che i linguaggi ricorsivamente enumerabili (RE), ovvero quelli accettati dalle MT, si suddividono in due classi:

- Decidibili: un qualsiasi algoritmo, sia in caso di successo che di fallimento, la MT si ferma
- Semidecidibili: in caso di successo la MT si ferma, altrimenti andrà avanti all'infinito

8.2.1 Linguaggi ricorsivi

Diremo "ricorsivo" un linguaggio L se L = L(M) per una qualche MT M tale che:



- Se ω è in L, allora M accetta (e si arresta)
- Se ω non è in L, allora M si arresta pur non entrando in uno stato accettante

Quindi abbiamo:

- Linguaggi ricorsivi
- Linguaggi ricorsivamente enumerabili ma non ricorsivi
- Linguaggi non ricorsivamente enumerabili

8.2.2 Complementi di linguaggi ricorsivi e RE

Teorema: Se L è RE e anche il suo complemento $\bar{}$L lo è, allora L è ricorsivo.

Teorema: se L è ricorsivo, anche $\bar{}$L lo è.

Perché si dice "ricorsivo" come sinonimo di "decidibile"? deriva dalla matematica precedente ai computer, quando era prassi servirsi di formalismi basati sulla ricorsione (e non sull'iterazione) a sostegno del concetto di computazione. Queste notazioni sono affini ai linguaggi funzionali. In questo senso "ricorsivo" significa "abbastanza semplice da poterlo risolvere con una funzione ricorsiva che ha sempre termine".

Il termine ricorsivamente enumerabile fa riferimento allo stesso campo concettuale: una funzione può elencare tutti gli elementi del linguaggio in un certo ordine, ovvero li enumera. I linguaggi i cui elementi possono essere elencati in un certo ordine sono gli stessi accettati da una MT.

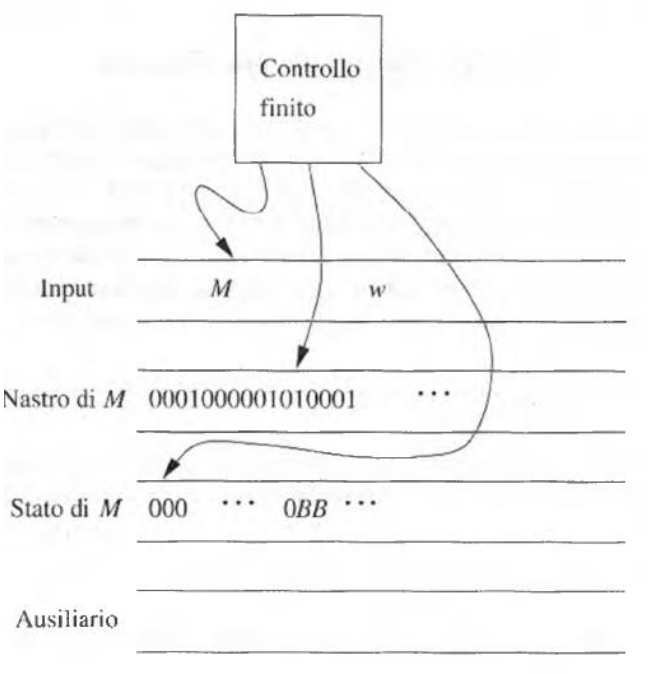
Possiamo riassumere i due teoremi come segue. Delle nove possibilità di collocare un linguaggio L e il suo complemento $\bar{L}$ all'interno del grafico di prima, solo quattro sono consentiti:

1) Sia L sia $\bar{L}$ sono ricorsivi, ovvero si trovano entrambi nel cerchio interno
2) Né L né $\bar{L}$ sono RE, cioè sono entrambi nel cerchio esterno
3) L è RE, $\bar{L}$ è non RE
4) L è non RE, $\bar{L}$ è RE

8.2.3 Il linguaggio universale L_u

Una singola MT può essere usata come un "computer a programma memorizzato", che legge un programma e i dati relativi da uno o più nastri su cui risiede l'input.

Definiamo L_u, il linguaggio universale, come l'insieme delle stringhe binarie che codificano una coppia (M, ω), dove M è una MT con alfabeto di input binario e ω è una stringa binaria tale che ω sia in L(M).



In altre parole, Lu è l'insieme delle stringhe che rappresentano una MT e un input da essa accettato. Mostreremo che esiste una MT U, detta macchina di Turing universale, tale che L_u = L(U).

Per semplicità descriviamo U come macchina di Turing multinastro:

- Primo nastro: contiene Il codice della MT assieme alla stringa ω
- Secondo nastro: contiene il nastro simulato di M, ricorrendo alla stessa forma binaria vista prima
- Terzo nastro: contiene lo stato di M, sempre codificato come visto prima

Ecco come opera U:

1) Esamina il codice di M per assicurarsi se sia valido, in caso contrario si arresta senza accettare
2) Prepara il secondo nastro con l'input
3) Scrive lo stato iniziale di M sul terzo nastro e muove la testina del secondo nastro verso la prima cella simulata
4) per simulare una transizione si usa la forma $0^i10^j10^k10^l10^m$ dove, come visto prima:
    a. 0^i: stato attuale
    b. 0^j: input
    c. 0^k: nuovo stato
    d. 0^l: valore sostituito
    e. 0^m: codice spostamento

5) Se M non ha transizioni per lo stato simulato e il simbolo di nastro, non ci sarà alcuna transizione
6) Se M entra nello stato accettante, allora U accetta

In questo modo U simula M su ω, U accetta la coppia codificata (M, ω) se e solo se M accetta ω.

8.2.4 Indecidibilità del linguaggio universale

Abbiamo individuato un problema che è RE ma non ricorsivo, il linguaggio L_u.

Ecco come lo possiamo dimostrare: supponiamo che L_u sia ricorsivo. Per il teorema visto prima, allora $\bar{L}_u$ è ricorsivo, quindi possiamo costruire una MT che accetta L_d.

Perché L_d? Perché possiamo ridurre il complemento di L_u a L_d, in quanto L_u accetta le coppie (M, ω) dove M accetta in input ω mentre L_d il contrario.

Sapendo che L_d è non ricorsivamente enumerabile, si entra in contraddizione con l'ipotesi di L_u ricorsivo.

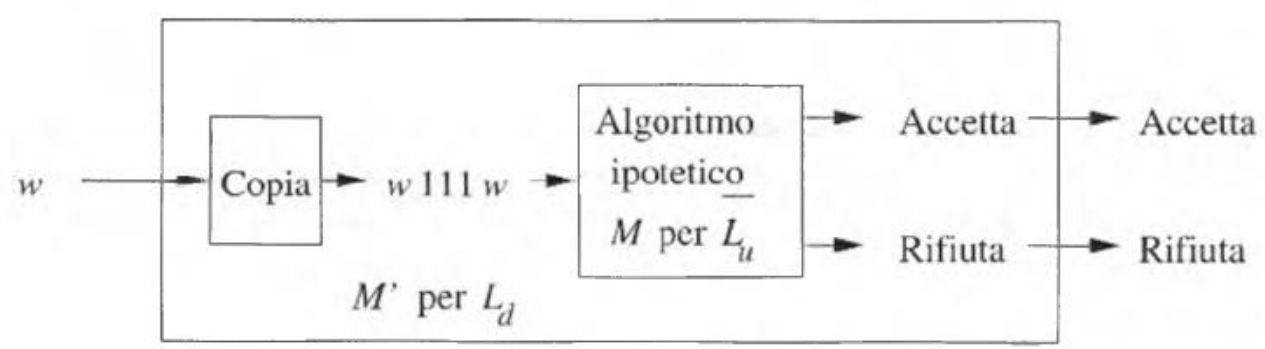
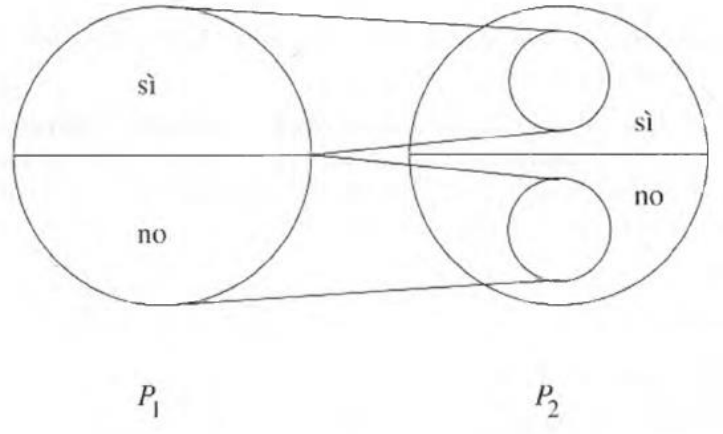


Figura 9.6 Riduzione di L_d a $\overline{L_u}$.

8.3 PROBLEMI INDECIDIBILI RELATIVI ALLE MACCHINE DI TURING

8.3.1 Riduzioni



Se abbiamo un algoritmo per convertire le istanze di un problema P_1 in istanze del problema P_2 che hanno la stessa risposta, allora diciamo che "P_1 si riduce a P_2".

Possiamo avvalerci di questa dimostrazione per provare che P_2 è "difficile" almeno quanto P_1, di conseguenza se P_1 non è ricorsivo, allora P_2 non può essere ricorsivo.

Se P_1 è non RE, allora P_2 non può essere RE.

In termini formali una riduzione da P_1 a P_2 è una macchina di Turing che riceve un'istanza di P_1 scritta sul nastro e si arresta con un'istanza di P_2 sul nastro.

Teorema: se esiste una riduzione da P_1 a P_2, allora:

- Se P_1 è indecidibile, allora lo è anche P_2
- Se P_1 è non RE, lo è anche P_2

Dimostrazione: supponiamo che P_1 sia indecidibile. Se è possibile decidere P_2, allora possiamo combinare la riduzione da P_1 a P_2 con l'algoritmo che decide P_2 per costruire un algoritmo che decide P_1.

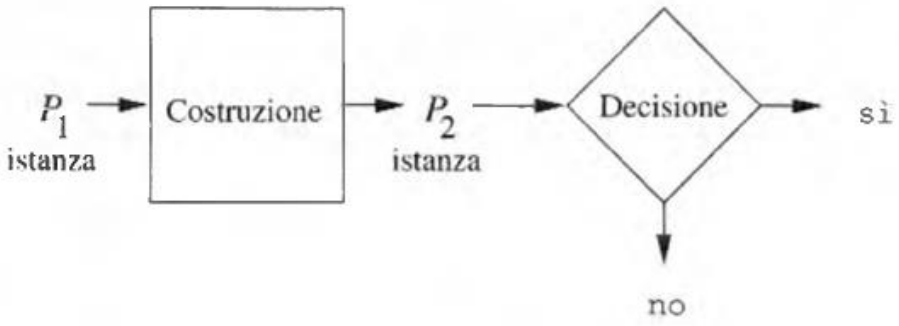


Figura 8.7 Se siamo in grado di risolvere il problema P_2, possiamo usare la soluzione per risolvere il problema P_1.

Supponiamo di avere un'istanza di P_1. Applichiamo a ω l'algoritmo che la converte in un'istanza x di P_2. Applichiamo poi a x l'algoritmo che decide P_2. Se l'algoritmo dice "sì", allora x è in P_2. Poiché abbiamo ridotto P_1 a P_2, sappiamo che la risposta a ω per P_1 è "sì", ossia ω è in P_1. Analogamente il contrario, quindi la risposta a "x è in P_2?" è uguale a "ω è in P_1?".

Abbiamo contraddetto l'ipotesi che P1 sia indecidibile, quindi o anche P2 è indecidibile oppure P1 è decidibile.

Ora supponiamo la seconda parte, ovvero che se P1 è non RE, lo è anche P2.

Diponiamo di un algoritmo che riduce P1 a P2, ma abbiamo solo una procedura per riconoscere P2: in altre parole esiste una MT che dice "sì" se il suo input è in P2, ma può non arrestarsi in caso contrario. Come prima riduciamo P1 a P2, applichiamo poi a x la MT per P2. Se x è accettato, accettiamo ω.

8.3.2 Macchine di Turing che accettano il linguaggio vuoto

Se $L(M_i) = \emptyset$, ossia M_i non accetta alcun input, allora ω è in L_ε. Quindi L_ε è il linguaggio formato da tutte le MT codificate il cui linguaggio è vuoto. Il suo complemento è $L_{n\varepsilon}$, ovvero l'insieme delle MT che accettano almeno una stringa di input.

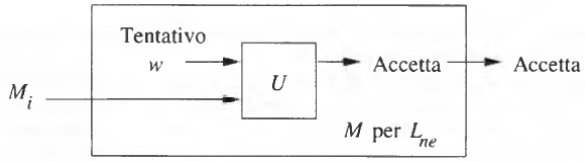


Figura 9.8 Costruzione di una NTM che accetta L_{ne}.

Teorema: $L_{n\varepsilon}$ è RE.

Dimostrazione: il modo più semplice è pensare che alla MT basta trovare una sola stringa accettata, il problema è che se $L(M) = \emptyset$, allora andrà avanti all'infinito cercando una stringa valida. Quindi è RE.

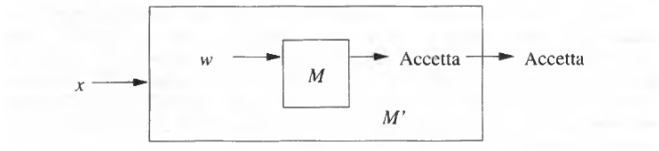


Figura 9.9 Schema della TM M' costruita da (M, w) nel Teorema 9.9. M' accetta un input arbitrario se e solo se M accetta w.

Teorema: $L_{n\varepsilon}$ non è ricorsivo.

Dimostrazione: c'è un algoritmo per ridurre L_u a $L_{n\varepsilon}$.

8.3.3 Il teorema di Rice e la proprietà dei linguaggi RE

Il fatto che linguaggi come L_ε e $L_{n\varepsilon}$ siano indecidibili è un caso speciale di un teorema più generale: tutte le proprietà non banali dei linguaggi RE sono indecidibili, nel senso che è impossibile riconoscere per mezzo di una macchina di Turing le stringhe binarie che rappresentano codici di una MT il cui linguaggio soddisfa la proprietà.

Un esempio di proprietà dei linguaggi RE è "il linguaggio è libero dal contesto".

Una proprietà dei linguaggi RE è semplicemente un insieme di linguaggi RE. **Una proprietà è banale se è vuota Ø (ossia non viene soddisfatta da nessun linguaggio) o comprende tutti i linguaggi RE**. Altrimenti è non banale.

Teorema di Rice: ogni proprietà non banale dei linguaggi RE è indecidibile.

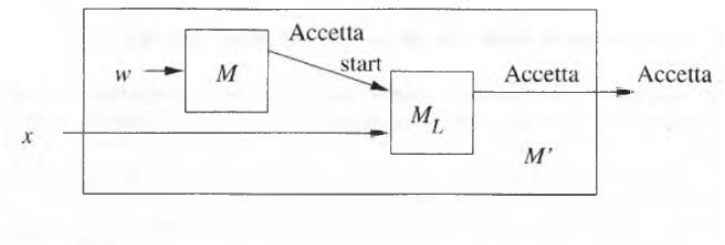


Figura 9.10 Costruzione di M' per la dimostrazione del teorema di Rice.

Dimostrazione: sia P una proprietà non banale dei linguaggi RE. Supponiamo che Ø, il linguaggio vuoto, non sia in P. dato che P è non banale, deve esistere un linguaggio non vuoto L che sia in P. sia Ml una MT che accetta L.

Ridurremo L_u a L_P dimostrando che L_P è indecidibile, dal momento che L_u è indecidibile.

L'algoritmo riceve una coppia (M, ω) e produce una TM M'. L(M') è Ø se M non accetta ω, e L(M') = L se M accetta ω.

M' è una MT a due nastri, simile alla MT universale ma la simulazione di M su ω è incorporata in M', quindi la seconda MT non deve leggere le transizioni di M sui suoi nastri.

Se necessario l'altro nastro di M' viene usato per simulare M_L sull'input x di M'.

La costruzione di M' da M e ω può essere realizzata da un algoritmo e dato che tale algoritmo trasforma (M, ω) in una M' che è in L_P se e solo se (M, ω) è in L_u, esso è una riduzione di L_u a L_p, e dimostra che la proprietà P è indecidibile.

Ci resta da considerare il caso in cui Ø è in P. esaminiamo allora la proprieta complemento $\bar{P}$, l'insieme dei linguaggi RE che non hanno la proprietà P. per quanto abbiamo visto sopra $\bar{P}$ è indecidibile.

8.3.4 Problemi sulle specifiche di macchine di Turing

Per il teorema di Riece tutti i problemi sulle macchine di Turing che toccano solo i linguaggi accettati sono indecidibili, ad esempio:

- Il linguaggio accettato da una MT è vuoto?
- Il linguaggio accettato da una MT è finito?
- Il linguaggio accettato da una MT è un linguaggio regolare?
- Il linguaggio accettato da una MT è un linguaggio libero dal contesto?

9 APPUNTI DI LABORATORIO

9.1 UNA BREVE INTRODUZIONE

9.1.1 Parser e Lexer

Un parser è un analizzatore sintattico ed è uno dei componenti di base di compilatori e traduttori. Analizza un testo per determinare se la sua struttura grammaticale rispetta una certa grammatica formale.

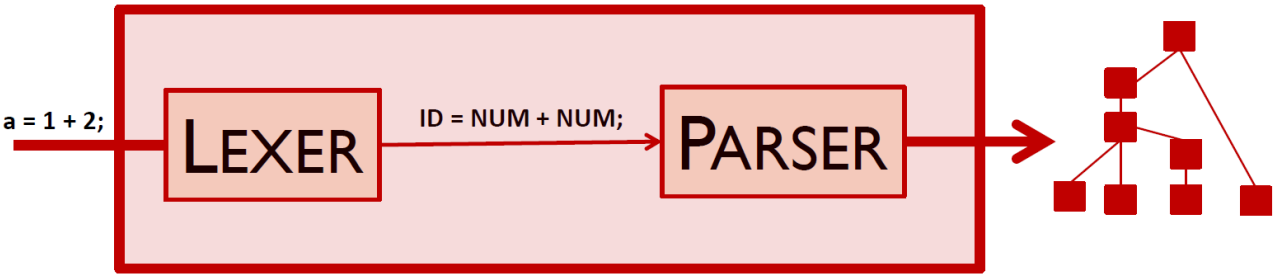
Verifica la correttezza sintattica, ovvero contribuisce all'identificazione degli errori di sintassi e produce un albero sintattico. Il testo è composto da sequenze di "token".

I parser ricorrono agli analizzatori lessicali (lexer) per identificare le sequenze di token a partire da una sequenza di caratteri. Spesso gli analizzatori lessicali sono tool separati rispetto ai parser.

Parser	Lexer
•Determina se la struttura grammaticale rispetta una certa grammatica formale	•Identifica le sequenze di token da sequenze di caratteri

possono essere creati manualmente, come qualsiasi altro software. Più spesso vengono creati attraverso tool semi-automatici:

- Generatori di parser: ad esempio Yacc
- Generatori di lexer: ad esempio LeX

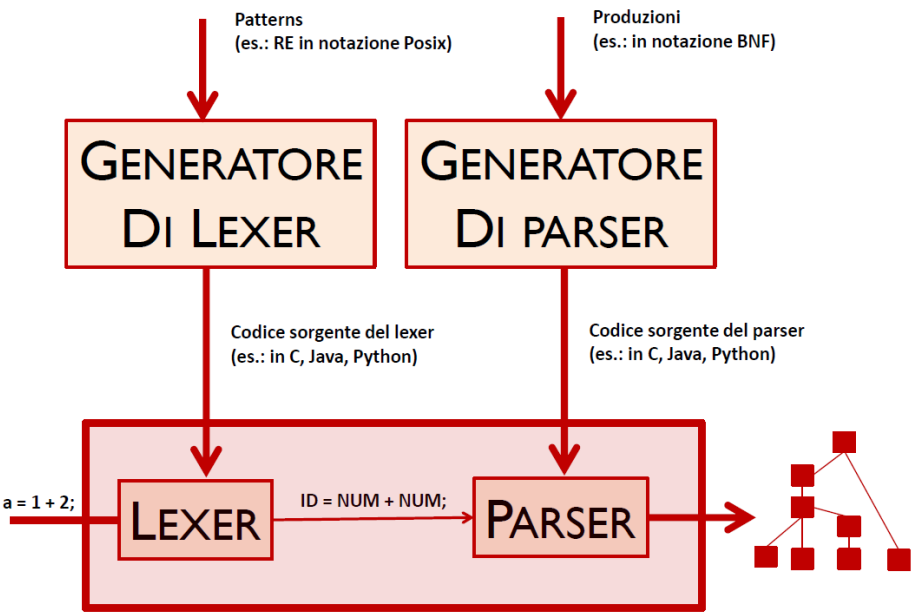


Un generatore di parser (compiler generator) genera un programma che implementa un automa a pila. La specifica di un parser generator (il suo input) contiene sempre tre sezioni:

- Dichiarazioni
- Regole di traduzione
- Codice user-defined

Un generatore di lexer invece fa sì che ogni pattern p_i avrà associata una "action" $\{a_i\}$. Tipicamente un'azione è un frammento di programma che ritorna un token che rappresenta una stringa "riconosciuta" da p_i. Lex genera un programma che implementa un automa a stati finiti. La specifica di Lex contiene sempre tre sezioni:

- Codice user-defined
- Definizioni
- Regole di traduzione



Questo programma è il lexer, chiamato dal parser un token alla volta. Esso analizza l'input (disponibile), un carattere alla volta, finchè non trova il prefisso più lungo che ha un match con p_i, quindi esegue a_i finchè non torna il controllo al parser (insieme al token identificato).

Lexer

- Trasforma una sequenza di input in una serie di token, ovvero di identificatori.
- a = 1 + 2; --> ID = NUM + NUM;
- Yacc

Parser

- Analizza la sequenza di token generata dal lexer e determina se la sua struttura formale rispetta una grammatica formale
- ID = NUM + NUM --> albero sintattico
- JFlex

9.2 PARSER: YACC E JAVA

BYACC/J usa una sintassi compatibile con quella di Yacc. Può generare (anche) codice sorgente in linguaggio Java invece che solo in C++.

Per eseguire BYACC/J dalla riga di comando e produrre codice sorgente in linguaggio Java bisogna scrivere, dopo essersi posizionati nella cartella bin all'interno della cartella di installazione di BYACC/J.:

```
yacc -J <input_file>
```

9.2.1 Struttura dei file .y in BYACC/J

Dichiarazioni	Opzionale	Minimo contenuto valido:
%%	Obbligatorio	%%
Regole di traduzione	Opzionale	La struttura è uguale a quella di Yacc originale.
%%	Opzionale	
Codice user-defined	Opzionale	

9.2.2 Definizione della grammatica

- Simboli non terminali
  - Convenzione: scritti in minuscolo
  - Esempio: exp, stmt, …
- Simboli terminali (token)
  - Convenzione: scritti in maiuscolo
  - Esempio: INTEGER, FLOAT, IF, WHILE, ';', …
- Regole sintattiche
  - Chiamate produzioni
  - Esempio: exp: exp '+' exp | exp '*' exp;

9.2.3 Definizione dei simboli

Dichiarazioni

%%

Regole di traduzione

%%

Codice user-defined

Si definiscono nella sezione delle dichiarazioni:

- Non terminali: %type [nome], ad esempio "%type string_ass"
- Terminali: %token [NOME], ad esempio "%token NUM"

Per default tutti i simboli sono di tipo int, ma è possibile utilizzare altri tipi di dato:

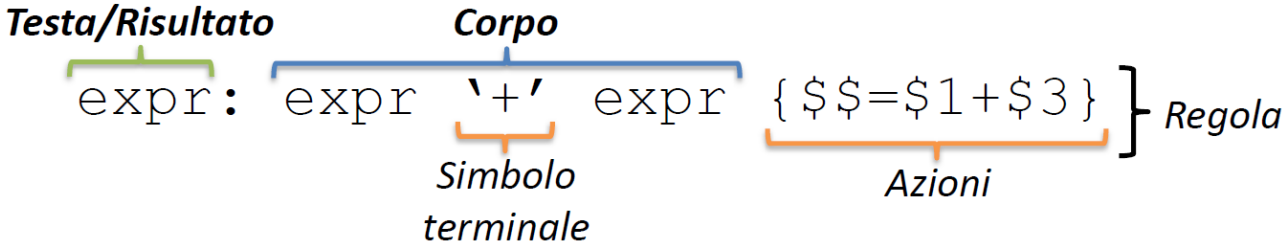
- La classe ParserVal fornisce dei metodi che permettono l'uso di tipi di dato differenti da int
- Associando il tipo ai simboli terminali e non terminali attraverso l'uso di parentesi angolari nei costrutti che ne permettono la dichiarazione, ad esempio:
  - %token<dval> NUM_DOUBLE
  - %token<ival> NUM_INT
  - %type<sval> stringa
  - %type<oval> oggetto

9.2.4 Definizione delle regole

Forma:

```
testa:      corpo       azioni
```

- Testa: simbolo non terminale a cui si riduce l'espressione nella parte destra della regola
- Corpo: è composto da simboli terminali e non terminali
- Azioni: insieme di istruzioni Java che vengono eseguite se il corpo della produzione è verificato



È possibile elencare "parti destre" alternative, che portano alla riscrittura dello stesso simbolo non terminale nella parte sinistra.

```
expr:  expr '+' expr {$$ = $1 + $3} |
       expr '*' expr {$$ = $1 * $3};
```

```
expr:  /* stringa vuota */ |
       expr_1
```

se la parte destra è vuota, la produzione viene soddisfatta anche dalla stringa vuota

```
expr_seq:  expr |
           expr_seq ',' expr;
```

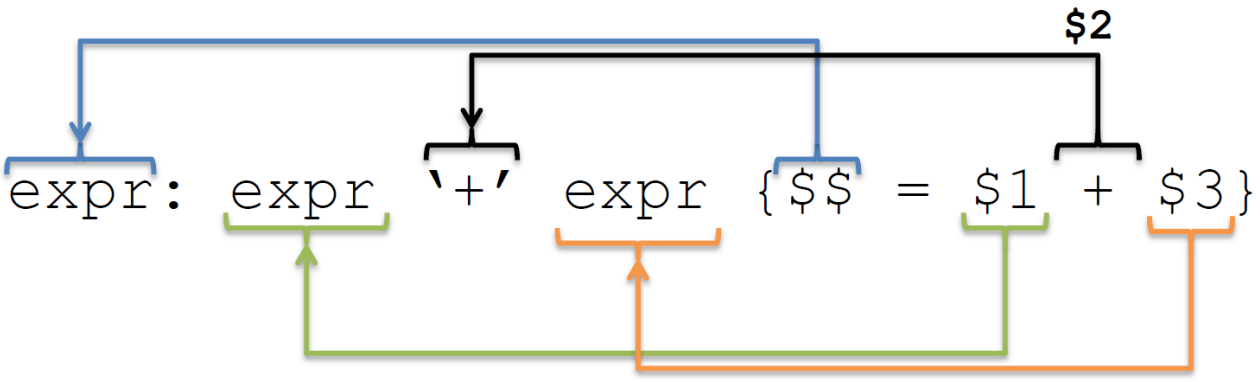
la produzione è ricorsiva se il simbolo non terminale della parte sinistra compare anche nella parte destra

9.2.5 Produzioni e azioni

Un'azione è una parte di codice Java che viene eseguita quando la produzione viene applicata (riduzione)

Il valore dell'n-esimo elemento della produzione corrisponde a $n; $$ rappresenta la parte sinistra; se non si specifica nessuna azione, per default si ha $$ = $1.

Il tipo associato a $n è quello dell'elemento corrispondente, si può forzare un altro tipo usando la sintassi $<tipo>n.



9.2.6 Associativa e precedenza

È possibile definire l'associatività dei simboli a sinistra o a destra:

Associatività a sinistra	%left OP	x OP y OP z	⇒	(x OP y) OP z
Associatività a destra	%right OP	x OP y OP z	⇒	x OP (y OP z)
Non associativa	%nonassoc OP	x OP y OP z	⇒	Errore di sintassi

		Priorità
%right	'='	+
%left	'+' '-'	
%left	'*' '/'	-

È possibile definire anche la precedenza, attraverso l'ordine delle dichiarazioni

9.2.7 Codice user-defined

È necessario definire due metodi:

- Void yyerror(String msg): permette di definire come trattare i messaggi di errore
- Int yylex(): usato dal parser per ottenere i token identificati dal lexer

È possibile definire altri metodi e campi, che verranno inclusi nella classe generata:

- Costruttori personalizzati della classe Parser
- Variabili di supporto
- Il metodo main (che deve invocare il metodo yyparse() della classe Parser)

```
%{
import java.lang.Math;
import java.io.*;
import java.util.StringTokenizer;
%}

/* YACC Declarations */

%token NUM
%left '-' '+'
%left '*' '/'
%left NEG /* negation--unary minus */
%right '^' /* exponentiation */
```

Tutto ciò che è tra %{ e %} viene ignorato da BYACC/J. Quindi può essere usato per inserire definizioni Java, include e import di librerie. Qua accanto un esempio.

9.2.8 Come eseguire BYACC/J: parametri

Obbligatori:

- -J: deve essere specificato esplicitamente per fare generare codice Java a BYACC/J
- <input_file>: contiene la definizione della grammatica

Opzionali:

- -Jclass=<nome_classe>: permette di specificare il nome della classe generata (e del file .java che la contiene
- -Jpackage=<nome_package>: permette di specificare il package della classe generata
- -Jthrows=<lista_eccezioni>: permette di specificare quali eccezioni possono essere "rilanciate" dal metodo yyparse().

L'inclusione di almeno uno di questi parametri rende non necessario l'uso del parametro obbligatorio -J.

9.2.9 Come eseguire BYACC/J: esempio d'uso

Per creare un file: `yacc -J esempio.y`

Compilazione di tutte le classi Java nella directory di installazione di BYACC/J: `javac *.java`

9.3 FLEX E JAVA

Jflex – The Fast Scanner Generator for Java.

È compatibile con la sintassi di LeX, è scritto in Java e produce codice Java.

Per eseguire Jflex dalla riga di comando: `jflex <options> <inputfiles>`

Oppure `java JFlex.Main <options> <inputfiles>`

9.3.1 Struttura dei file .flex

Differisce leggermente dalla struttura dei file .lex per i tool LeX. Presenta le stesse tre sezioni di dichiarazione, ma in un ordine diverso (il codice user defined è in testa al file).

`Codice user-defined`	Opzionale	Un file di input di JFlex contenente sono %% genera un lexer che copia tutto il testo preso in input sull'output.
`%%`	Opzionale	
`definizioni`	Opzionale	
`%%`	Obbligatorio	
`Regole di traduzione`	Opzionale	

9.3.2 Codice user-defined

Viene copiato in blocco all'inizio del file .java generato. In questa sezione vengono dichiarate le direttive package e import.

Si possono definire anche classi di supporto (helper classes) ma non è considerata buona pratica, meglio definirle in file .java esterni.

9.3.3 Definizioni

Permettono di personalizzare la classe generata, devono obbligatoriamente iniziare con il carattere %.

Opzioni della classe:

- %class "nome classe"
- %extends "nome classe"
  e
  %implements "nome interfaccia", …
- %public, %final e %abstract

Codice della classe:

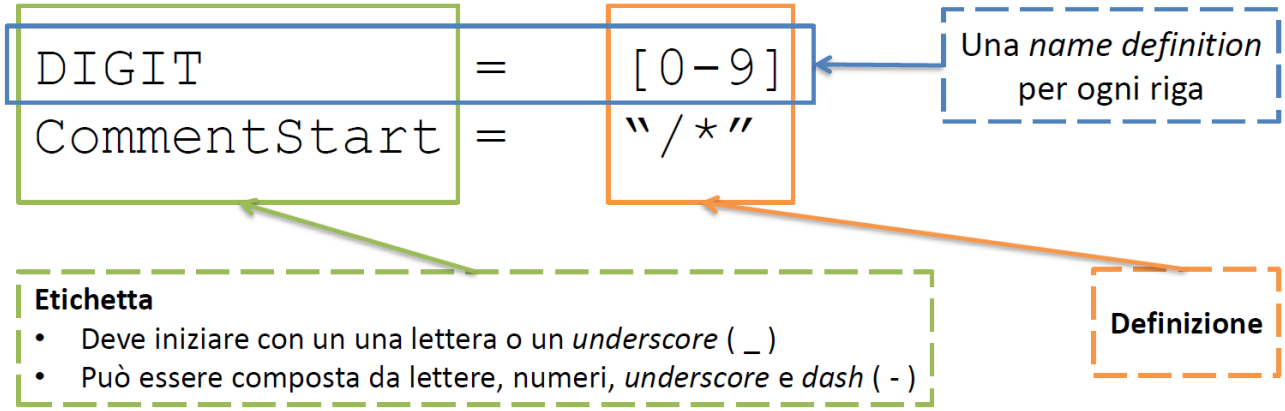
`%{`
…
`%}`
	Codice incluso in blocco all'interno della classe, ad esempio le dichiarazioni dei campi della classe. Se vi sono più dichiarazioni di questo tipo vengono concatenate

`%init{`
…
`%init}`
	Codice incluso in blocco all'interno del costruttore della classe, ad esempio el dichiarazioni dei campi delal classe. Se vi sono più dichiarazioni di questo tipo, vengono concatenate.

Compatibilità con BYACC/J: %byaccj

9.3.4 Definizioni: name definitions

Chiamate anche macro, servono per definire una volta sola i pattern che si ripetono più volte, anche in diverse regole lessicali.



Si referenziano attraverso la notazione {etichetta}, ad esempio "{DIGIT}" equivale a [0-9].

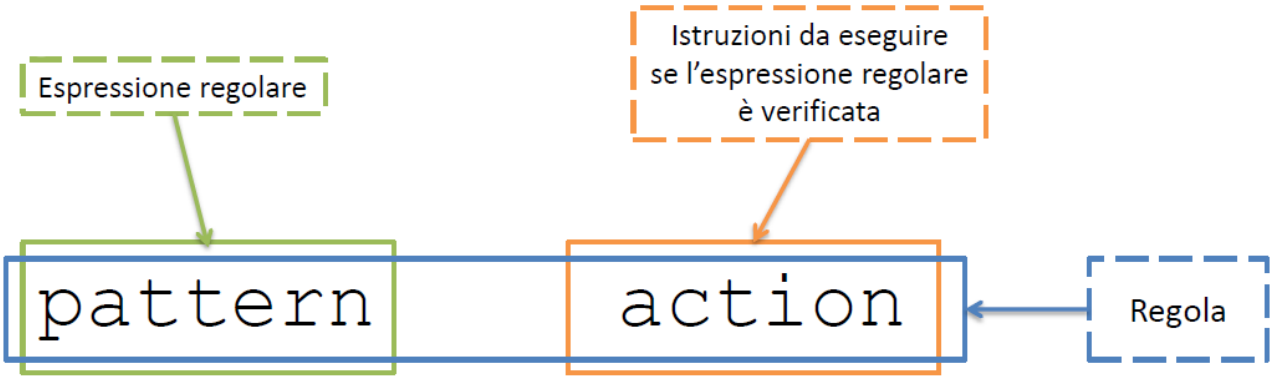
Ad esempio:

… `DIGIT = [0-9]` …	Definzioni

`{DIGIT}+"."{DIGIT}* { System.out.println(…);}`
Che equivale a
`[0-9]+"."[0-9]* {System.out.println(…);}`
	Regole

9.3.5 Regole di traduzione lessicale

Identificano ed eventualmente restituiscono i token.



Strategia di risoluzione dei pattern:

1) Viene scelto il più lungo tra i pattern verificati
2) Se più pattern con la stessa lunghezza sono verificati, allora viene selezionato il primo
3) Se nessun pattern è verificato, il comportamento di default è di copiare il carattere seguente sullo standard output

9.3.6 Definizioni: start conditions

È possibile "pilotare" la valutazione dei pattern attraverso le start conditions, ovvero gli stati di una macchina a stati finiti.

Due tipi:

- Inclusive
- Exclusive

Nella sezione delle dichiarazioni, si dichiarano nella forma:

- Inclusiva: `%s list_of_inclusive_start_conditions`
- Esclusiva: `%x list_of_exclusive_start_conditions`

Ad esempio: `%s COND1, COND2, COND3, …`

In alternativa possono essere dichiarate con:

- Inclusiva: `%state list_of_inclusive_start_conditions`
- Esclusiva: `%xstate list_of_exclusive_start_conditions`

Esiste uno stato predefinito YYINITIAL dichiarato implicitamente.

Nella sezione delle regole, si referenziano attraverso l'operatore <nomi_start_conditions> anteposto al pattern di una regola, es:

```
<COND2, COND3>[0-9]* {System.out.println(…);}
```

In alternativa, è possibile associare più regole lessicali allo stesso stato attraverso la sintassi compatta, ad esempio:

```
<COND1, COND2> {
      [0-9]* {System.out.println(…);}
      [a-z]+ {System.out.print (…);}
}
```

Una regola può avere una o più start conditions oppure nessuna. Una regola senzaq start conditions è associata implicitamente allo stato YYINITIAL.

Le regole che non specificano alcuna start condition vengono utilizzate quando la condizione attiva (stato) è inclusive. YYINITIAL è uno stato inclusive.

Se la condizione attiva è exclusive, allora sono le regole a cui è associata esplicitamente vengono eseguite.

9.3.7 Classe generata: metodi principali

`int yylen()`	Invocato per processare la stringa di caratteri in input al lexer
`String yytext()`	Restituisce la stringa che verifica l'espressione regolare
`int yylength()`	Restituisce la lunghezza della stringa che verifica l'espressione regolare
`char yycharat(int pos)`	Restituisce il carattere in posizione `pos` nella stringa che verifica l'espressione regolare.

	È equivalente a `yytext().chatAt(pos)`, ma più veloce. `pos` deve essere compreso tra 0 e `yylength()-1`
`void yyclose()`	Chiude lo stream di input
`int yystate()`	Restituisce lo stato corrente del lexer
`void yybegin(int lexicalState)`	Pone il lexer nello stato lexicalState

9.3.8 Come eseguire JFlex: parametri

- Obbligatori
  - `<inputfiles>`: uno (o più) file .flex da cui generare il codice del lexer
    esempio: `file_1.flex file_2.flex … file_n.flex`
- Facoltativi
  - `-d <dir>`: memorizza i file .java in dir
  - `-verbose` o `-v`: mostra tutti i messaggi
  - `-quiet` o `-q`: mostra solo i messaggi di errore

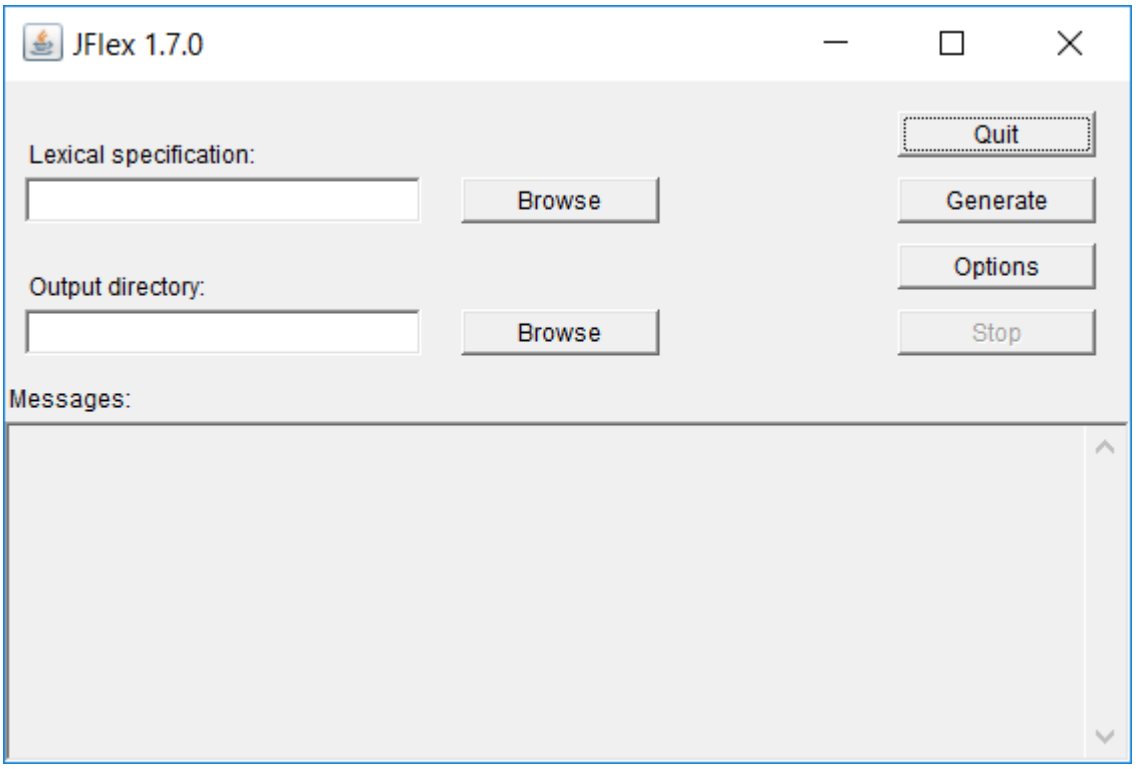
9.3.9 Come eseguire Jflex: esempio d'uso tramite prompt

- Creazione del file `esempio.flex`
- Esecuzione del comando `jflex -d C:\{cartella dove risiede esempio.flex} -verbose esempio.flex`
  oppure
  `java JFlex.Main -d C:\{cartella dove risiede esempio.flex} -v esempio.flex`
- Compilazione della cartella `C:\{cartella dove risiede esempio.flex}` con il "classico" `javac *.java`

9.3.10 Come eseguire Jflex: esempio d'uso tramite interfaccia

- Esecuzione del comando jflex

- Si apre l'interfaccia



9.3.11 Le espressioni regolari

I pettern delle regole di Jflex si basano sulle espressioni regolari per riconoscere le sequenze di caratteri in input (pattern matching).

Le espressioni regolari si basano sul formato standard utilizzato dai tool disponibili in ambiente POSIX.

9.3.12 Pattern semplici

Pattern	Descrizione
x	Il carattere x (minuscolo)
X	Il carattere X (maiuscolo)
[amz]	Il carattere a, m o z
[a-z]	Un carattere compreso tra a e z (estremi inclusi)
[A-Z]	Un carattere compreso tra A e Z (estremi inclusi)
[abd – zZ]	Un carattere miuscolo compreso tra a e z, escluso c, più Z (maiuscolo)
[a-zA-Z]	Un carattere compreso tra a e z o tra A e Z
.	Un carattere qualsiasi
\n	Andata a capo

9.3.13 Pattern complessi

Con r ed s sono indicati dei pattern (espressioni regolari) generici, ad esempio[0-9], [a-zA-Z], ecc.

Pattern	Descrizione
[^r]	Classe di caratteri negata: identifica un carattere che non corrisponde al pattern specificato dopo [^. Esempio: [^a-c] è verificato per qualsiasi carattere che NON sia a, b o c
r?	Il pattern può non comparrire o comparire una sola volta
r*	Il pattern può comparire zero o più volte
r+	Il pattern può comparire una o più volte

r{n}	Il pattern deve ripetersi n volte
r{n, m}	Il pattern può ripetersi da minimo n volte a massimo m volte
r{n, }	Il pattern deve ripetersi minimo n volte, nessun limite superiore
rs	Il pattern r deve essere seguito dal pattern s
r|s	Il pattern r oppure il pattern s
^r	Il pattern r viene "cercato" solo nella parte iniziale della stringa di caratteri
(r)	Crea un gruppo contenente il pattern r

9.3.14 Caratteri di escape

Alcuni caratteri hanno un significato speciale nelle espressioni regolari, altri nel testo fornito in input al lexer. Non possono essere "usati" direttamente, bisogna effettuare l'escape, ovvero si antepone il carattere \ al carattere necessario

Pattern	Descrizione
\a, \b, \f, \n, \r, \t, \v	I caratteri speciali nelle stringe ANSI-C
***, \", \', \+, \?, \[, …**	I caratteri riservati delle espressioni regolari
\123, \x2a	Il carattere con valore ottale 123 e il carattere con valore esadecimale 2a

9.3.15 Estensioni di JFlex

I pattern di Lex possono essere definiti usando anche costrutti "non standard".

Pattern	Descrizione
{nome_definizione}	Richiama una name definition e la include nel pattern
r/s	Il pattern (composto) è verificato se r segue il patter s, ma solo la stringa che verifica r viene considerata l'input corrente
"[a]\"foo"	Considera tutti i caratteri racchiusi tra " come un pattern letterale

9.3.16 Esempi di pattern

Pattern	Stringa valida	Stringa NON valida
[0-9]+	123 A123	aaabaa
[a-z]+	prova	123
[A-Z]?	A 1° A123 1	
^[A-Z]+	A A1 AA AA1	1a
[A-Z]+$	1° 1AA	A1 1a
a{3}b	aaaabaa	Aabba
a{2, 4}b	aab aaab	abaa
[a-z]+[\t]+[0-9]+	prova 1	123

	Prova 1	prova1
Esempi reali		
(19|20)[0-9]{2} Anni dal 1900 al 2099	1991 2012	12 2100
0[1-9]|1[012] Mesi in formato numerico a due cifre	12	1
(19|20)[0-9]{2}\\(0[1-9]|1[012])\\(0[1-9]|[12][0-9]|3[01]) Data in formato yyyy\mm\gg	2012\12\12	2012-12-12
^[a-zA-Z0-9-_\.]+@[a-zA-Z0-9-]+\.[a-zA-Z]{2,4}$	nome.c@e.com	nome.c@e

9.4 BYACC/J E JFLEX

9.4.1 Esecuzione dei tool

- Aprire il prompt
- Portarsi nella cartella %\jflex-[ver]\bin
- Eseguire in sequenza i due tool
  - yacc -J [file.y]
  - jflex [file.flex]
    vengono così generati i file .java
- eseguire il compilatore Java
  - javac *.java
- eseguire il programma generato
  - java -cp . Parser

9.5 ESEMPIO: UNA CALCOLATRICE SCRITTA CON BYACC/J E JFLEX

File calc.flex

```
%%

%byaccj

%{
  private Parser yyparser;

  public Yylex(java.io.Reader r, Parser yyparser) {
    this(r);
    this.yyparser = yyparser;
  }
%}

NUM = [0-9]+ ("." [0-9]+)?
NL  = \n | \r | \r\n

%%

/* operators */
"+" |
"-" |
"*" |
"/" |
"^" |
"(" |
")"    { return (int) yycharat(0); }

/* newline */
{NL}   { return Parser.NL; }

/* float */
{NUM}  { yyparser.yylval = new ParserVal(Double.parseDouble(yytext()));
         return Parser.NUM; }

/* whitespace */
[ \t]+ { }

\b     { System.err.println("Sorry, backspace doesn't work"); }

/* error fallback */
[^]    { System.err.println("Error: unexpected character '"+yytext()+"'"); return -1;
```

File calc.y

```
%{
  import java.io.*;
%}

%token NL          /* newline  */
%token <dval> NUM  /* a number */

%type <dval> exp

%left '-' '+'
%left '*' '/'
%left NEG          /* negation--unary minus */
%right '^'         /* exponentiation        */

%%

input:   /* empty string */
       | input line
       ;

line:    NL      { if (interactive) System.out.print("Expression: ")
       | exp NL  { System.out.println(" = " + $1);
                   if (interactive) System.out.print("Expression: ")
       ;

exp:     NUM                { $$ = $1; }
       | exp '+' exp        { $$ = $1 + $3; }
       | exp '-' exp        { $$ = $1 - $3; }
       | exp '*' exp        { $$ = $1 * $3; }
       | exp '/' exp        { $$ = $1 / $3; }
       | '-' exp  %prec NEG { $$ = -$2; }
       | exp '^' exp        { $$ = Math.pow($1, $3); }
       | '(' exp ')'        { $$ = $2; }
       ;

%%

  private Yylex lexer;


  private int yylex () {
    int yyl_return = -1;
    try {
```

```
    yylval = new ParserVal(0);
    yyl_return = lexer.yylex();
  }
  catch (IOException e) {
    System.err.println("IO error :"+e);
  }
  return yyl_return;
}


public void yyerror (String error) {
  System.err.println ("Error: " + error);
}


public Parser(Reader r) {
  lexer = new Yylex(r, this);
}


static boolean interactive;

public static void main(String args[]) throws IOException {
  System.out.println("BYACC/Java with JFlex Calculator Demo");

  Parser yyparser;
  if ( args.length > 0 ) {
    // parse a file
    yyparser = new Parser(new FileReader(args[0]));
  }
  else {
    // interactive mode
    System.out.println("[Quit with CTRL-D]");
    System.out.print("Expression: ");
    interactive = true;
    yyparser = new Parser(new InputStreamReader(System.in));
  }

  yyparser.yyparse();

  if (interactive) {
    System.out.println();
    System.out.println("Have a nice day");
  }
}
```

APPENDICE A: ALFABETO GRECO

Lettere dell'alfabeto greco

1	α A	alfa	9	ι I	iota	17	ρ P	rho
2	β B	beta	10	κ K	cappa	18	σ Σ	sigma
3	γ Γ	gamma	11	λ Λ	lambda	19	τ T	tau
4	δ Δ	delta	12	μ M	mi	20	υ Y	ipsilon
5	ε E	epsilon	13	ν N	ni	21	φ Φ	fi
6	ζ Z	zeta	14	ξ Ξ	csi	22	χ X	chi
7	η H	eta	15	ο O	omicron	23	ψ Ψ	psi
8	θ Θ	theta	16	π Π	pi	24	ω Ω	omega