FONDAMENTI, LINGUAGGI E TRADUTTORI

Alessandro Zappatore

Università del Piemonte Orientale
Anno accademico 2024/2025, 1° semestre

1 Alfabeto e linguaggio

Un *alfabeto* è un insieme finito di elementi chiamati *simboli terminali* o *caratteri*. Ad esempio

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

è un alfabeto composto da 3 elementi a, b, c (la sua cardinalità è 3).

Una *stringa* (o *parola*) è una sequenza, ovvero un insieme ordinato eventualmente con ripetizioni, di caratteri.

Un *linguaggio* è un insieme di stringhe di un alfabeto specifico. Dato un linguaggio, una stringa che gli appartiene è detta *frase*. Ad esempio, possiamo definire un linguaggio L

$$L = \{a, ab, bc, cccc\}$$

le cui parole al suo interno sono formate esclusivamente delle lettere dell'alfabeto specificato in precedenza.

La *cardinalità* di un linguaggio è definita dal numero di frasi che contiene. Se la cardinalità è finita, il linguaggio si dice *finito*.

Un linguaggio finito è una collezione di parole, solitamente chiamate *vocabolario*. Il linguaggio che non contiene frasi è chiamato *insieme vuoto* o *linguaggio* $\emptyset$.

La *lunghezza* $|x|$ di una stringa x è il numero di caratteri che contiene.

1.1 Operazioni sulle stringhe

Stringa vuota

La *stringa vuota* (o *nulla*), denotata con ε, soddisfa l'identità:

$$x \cdot \varepsilon = \varepsilon \cdot x = x$$

La stringa vuota non deve essere confusa con l'insieme vuoto; infatti, l'insieme vuoto è un linguaggio che non contiene stringhe, mentre l'insieme $\{\varepsilon\}$ ne contiene una, la stringa vuota.

Sottostringa

Sia la stringa $x = uyv$ il prodotto della concatenazione delle stringhe u, y e v: le stringhe u, y e v sono *sottostringhe* di x. In questo caso, la stringa u è un *prefisso* di x e la stringa v è un *suffisso* di x. Una sottostringa non vuota è detta *propria* se non coincide con x.

Concatenazione

Date le stringhe

$$x = a_1 a_2 \dots a_h \qquad\qquad y = b_1 b_2 \dots b_k$$

la *concatenazione*, indicata con '$\cdot$', è definita come:

$$x \cdot y = a_1 a_2 \dots a_h b_1 b_2 \dots b_k$$

La concatenazione non è commutativa, ma è associativa.

Inversione di stringa

L'*inverso* di una stringa $x = a_1 a_2 \dots a_h$ è la stringa $x^R = a_h a_{h-1} \dots a_1$.

Ripetizione

La potenza m-esima x^m di una stringa x è la concatenazione di x con se stessa per $m-1$ volte. Esempi:

$$x = ab \qquad x^0 = \varepsilon \qquad x^2 = (ab)^2 = abab$$

1.2 Operazioni sul linguaggio

Linguaggio neutro

Il linguaggio contenente esclusivamente la stringa vuota è detto *linguaggio neutro*. Ha cardinalità pari a 1.

$$L_N = \{\varepsilon\}$$
$$L \cdot L_N = L_N \cdot L = L$$

Linguaggio vuoto

Il linguaggio vuoto non contiene alcuna stringa, quindi la sua cardinalità è 0. Si indica con $\emptyset$.

$$L \cdot \emptyset = \emptyset \cdot L = \emptyset$$

Concatenazione

La concatenazione tra due linguaggi è il prodotto cartesiano tra le stringhe di entrambi i linguaggi. Ad esempio, dati i linguaggi L_1 e L_2

$$L1 = \{a, b, c\} \qquad L_2 = \{bb, cc\}$$

concatenandoli si ottiene

$$L_3 = L_1 \cdot L_2 = \{abb, acc, bbb, bcc, cbb, ccc\}$$

$|L_3| \leq |L_1| * |L_2|$

Inversione

L'inverso L^R di un linguaggio L è l'insieme delle stringhe che sono l'inverso di una frase di L.

Ripetizione

Come per le stringhe, è possibile l'elevamento a potenza.

$$L^m = L^{m-1} \cdot L \text{ per } m \geq 1 \qquad L^0 = \{\varepsilon\}$$

1.3 Operazioni sugli insiemi

Dato che un linguaggio è un insieme, si possono usare gli operatori unione '$\cup$', intersezione '$\cap$' e differenza '$\setminus$'. Sono applicabili inoltre le relazioni di inclusione '$\subseteq$', inclusione stretta '$\subset$' ed uguaglianza '$=$'.

Il *linguaggio universale* è l'insieme di tutte le stringhe, su un alfabeto Σ, di ogni lunghezza inclusa 0. Il linguaggio universale è infinito.

$$L_{universale} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots = \Sigma^*$$

Il *complemento* di un linguaggio L su un alfabeto Σ, denotato con $\neg L$, è la differenza insiemistica

$$\neg L = L_{universale} \setminus L$$

1.4 Operatore di Kleene e croce

Per definire linguaggi infiniti, si usano due operatori: l'operatore di Kleene '$*$' e l'operatore croce '$+$'.

Operatore di Kleene

Questa operazione è definita come unione di tutte le potenze del linguaggio base:

$$L^* = \bigcup_{h=0}^{\infty} L^h = L^0 \cup L^1 \cup L^2 \cup \dots$$

Può generare un numero infinito di parole composte da un numero infinito di caratteri.

Operatore croce

Questo operatore è derivato da quello precedente:

$$L^+ = \bigcup_{h=1}^{\infty} L^h = L^1 \cup L^2 \cup L^3 \cup \dots$$

2 Linguaggi regolari

2.1 Definizione di espressione regolare

Un linguaggio su un alfabeto $\Sigma = \{a_1, a_2, \dots, a_n\}$ è *regolare* se può essere espresso applicando finite volte le operazioni di concatenazione, unione e Kleene, a partire dai linguaggi unitari $\{a_1\}, \{a_2\}, \dots, \{a_n\}$ o la stringa vuota ε.

Più precisamente, un'*espressione regolare* è una stringa r contenente i caratteri terminali dell'alfabeto Σ e i metasimboli '$\cup$' (unione), '$\cdot$' (concatenazione), '$*$' (iterazione), 'ε' (stringa vuota) e parentesi, in accordo con le seguenti regole:

regola	significato
$r = \varepsilon$	stringa vuota
$r = a$	linguaggio unitario
$r = (s \cup t)$	unione di espressioni
$r = (s \cdot t)$	concatenazione di espressioni
$r = (s)^*$	iterazione di un'espressione

dove i simboli s e t sono espressioni regolari.

Esempio di espressione regolare

Proviamo a creare un linguaggio che generi tutti i numeri naturali, con o senza segno. L'alfabeto per questo linguaggio sarà:

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$$

Il primo simbolo della parola dovrà necessariamente essere '$+$' o '$-$', quindi la prima parte della nostra espressione regolare sarà:

$$(+ \cup - \cup \varepsilon)$$

I simboli a destra dovranno essere delle cifre, quindi:

$$(+ \cup - \cup \varepsilon)(1 \cup 9)(0 \cup 9)^*$$

In questo modo, il primo simbolo sarà almeno 1. Per poter generare anche lo 0:

$$(+ \cup - \cup \varepsilon)((1 \cup 9)(0 \cup 9)^*) \cup 0$$

2.2 Derivazioni

Formalizziamo il processo mediante il quale una data espressione regolare e produce il linguaggio in questione. Prendiamo in esame l'espressione regolare e_0:

$$e_0 = (((a \cup (bb))^*)((c^+) \cup (a \cup (bb))))$$

Questa espressione regolare è data dalla concatenazione delle due *sottoespressioni* e_1 ed e_2:

$$e_1 = ((a \cup (bb))^*) \qquad\qquad e_2 = ((c^+) \cup (a \cup (bb)))$$

La sottostringa s

$$s = (a \cup (bb))$$

è una sottoespressione di e_2 ma non di e_0.

Un operatore di unione o iterazione offre diversi modi per produrre stringhe. Effettuando una scelta, si può ottenere un'espressione regolare che definisce un linguaggio meno espressivo (in grado di generare meno parole), incluso in quello originale. Si dice che un'espressione regolare è una *scelta* di un'altra nei seguenti tre casi:

Derivazione da unione

Un'espressione regolare e_k, con $1 \leq k \leq m$ e $m \geq 2$, è una scelta dell'unione:

$$(e_1 \cup \cdots \cup e_k \cup \cdots \cup e_m)$$

Derivazione da * o +

Un'espressione regolare $e^m = e \ldots\ldots e$, con $m \geq 1$ è una scelta di e^* o e^+.

Derivazione da stringa vuota

La stringa vuota ε è una scelta di e^*.

Derivazione immediata

Si dice che un'espressione regolare e' *deriva* un'espressione regolare e'' ($e' \Rightarrow e''$) se una delle seguenti proposizioni è vera:

1. l'espressione regolare e'' è una scelta di e'
2. l'espressione regolare e' è la concatenazione di $m \geq 2$ sottoespressioni, e e'' è ottenuta da e' sostituendo una sottoespressione, ad esempio e'_k, con una scelta di e'_k, ad esempio e''_k:

$$\exists k, 1 \leq k \leq m \text{ tale che } e''_k \text{ è una scelta di } e'_k \wedge e'' = e'_1 \ldots e''_k \ldots e'_m$$

Tale derivazione è detta *immediata* in quanto effettua una sola scelta. Si dice che un'espressione regolare e_0 deriva un'espressione regolare e_n in $n \geq 1$ passi ($e_0 \Rightarrow^n e_n$) se le seguenti derivazioni immediate sono applicabili:

$$e_0 \Rightarrow e_1 \qquad\qquad e_1 \Rightarrow e_2 \qquad\qquad \ldots \qquad\qquad e_{n-1} \Rightarrow e_n$$

Esempi di derivazioni immediate:

$$a^* \cup b^+ \Rightarrow a^* \qquad\qquad a^* \cup b^+ \Rightarrow b^+ \qquad\qquad (a^* \cup bb)^* \Rightarrow (a^* \cup bb)(a^* \cup bb)$$

Le sottostringhe di un'espressione regolare sono scelte in ordine da più esterna a più interna.

2.3 Limiti dei linguaggi regolari

Le espressioni regolari, nonostante siano utili e pratiche, sono costrutti molto limitati. Ad esempio, l'espressione $a^n b^n$ con $n \geq 0$ non è formalizzabile come espressione regolare, in quanto $a^n b^n \neq a^* b^*$: infatti, il numero di a è vincolato al numero di b, non traducibile solo mediante operatori di espressioni regolari.

3 Grammatiche

Una *grammatica generativa* o *sintassi* è un insieme di semplici regole che possono essere applicate ripetutamente per generare tutte e sole le stringhe valide.

Esempio: palindromi

Il linguaggio L, descritto dall'alfabeto $\Sigma = \{a, b\}$, è definito, tramite l'operazione di inversione, come:

$$L = \{uu^R | u \in \Sigma^*\} = \{\varepsilon, aa, bb, abba, baab, \dots, abbbba, \dots\}$$

Contiene stringhe specularmente simmetriche. La grammatica G contiene quattro regole:
$L_p \to P \mid \varepsilon$
$P \to aPa$
$P \to bPb$
$P \to \varepsilon$
Per derivare le stringhe, basta rimpiazzare il simbolo P, detto *non terminale*, con la parte destra della regola di generazione, ad esempio:

$$P \Rightarrow aPa \Rightarrow abPba \Rightarrow abbPbba \Rightarrow \dots$$

Il processo di derivazione termina quando l'ultima stringa ottenuta non contiene più nessun carattere non terminale. Completiamo la derivazione:

$$abbPbba \Rightarrow abb\varepsilon bba = abbbba$$

Il linguaggio dei palindromi non è regolare.

3.1 Definizione di grammatica context-free

Una grammatica *context-free* (libera dal contesto) è una grammatica G, dove posso applicare una regola indipendentemente da quello che è avvenuto in precedenza, ed è definita da quattro entità:

- V *alfabeto dei non terminali*, un insieme di simboli non terminali
- Σ *alfabeto dei terminali*, un insieme di simboli terminali
- P insieme di *regole di produzione* (un linguaggio senza regole di produzione è un linguaggio vuoto)
- S un particolare non terminale detto *assioma*, dal quale parte la generazione ($S \in V$)

Una regola dell'insieme P è una coppia ordinata $X \to \alpha$, con $X \in V$ e $\alpha \in (V \cup \Sigma)^*$ $\quad V \times (V \cup \Sigma)^*$.

3.2 Derivazione e generazione

Siano $\beta, \gamma \in (V \cup \Sigma)^*$ e sia $\beta = \mu A \delta$ una stringa contenente un non terminale A, dove δ e μ sono stringhe, possibilmente vuote. Sia $A \to \alpha$ una regola della grammatica G e sia $\gamma = \mu\alpha\delta$ la stringa ottenuta rimpiazzando il non terminale A in β con la regola nella parte destra α. Questa relazione è detta *derivazione diretta*, da una forma sentenziale β a una forma sentenziale γ, si parla di forme sentenziali perché è presente un numero arbitrario di terminali e non terminali. Si dice che la stringa β deriva la stringa γ per la grammatica G, e scriviamo:

$$\beta \Rightarrow \gamma$$

La regola $A \to \alpha$ viene applicata in tale derivazione e la stringa α *riduce* al non terminale A. Il *linguaggio generato* o *definito* da una grammatica G, iniziando da un non terminale A, è l'insieme delle strignhe terminali che derivano da un non terminale A in uno o più passi:

$$L_A(G) = \{x \in \Sigma^* | A \Rightarrow^+ x\}$$

Se il non terminale è l'assioma S, si ha il linguaggio generato da G:

$$L(G) = L_S(G) = \{x \in \Sigma^* | S \Rightarrow^+ x\}$$

Un linguaggio è context-free se esiste una grammatica context-free che lo genera. Due grammatiche G e G' sono *equivalenti* se generano lo stesso linguaggio.

3.3 Grammatiche ridotte

Una grammatica G è detta *ridotta* se:

1. non sono presenti regole di produzione inutili (non torno alla stringa di partenza)
   $\forall A \in V \qquad \neg\exists A \xrightarrow{+} A$

2. qualsiasi non terminale A è *raggiungibile* dall'assioma
   $\forall A \in V \qquad L(A) \neq \Phi \qquad$ A assioma

3. qualsiasi non terminale A è *ben definito*, ovvero non genera un linguaggio vuoto
   $\forall A \in V \qquad S \xrightarrow{+} \alpha AB \qquad \alpha, \beta \in (\Sigma \cup V)^*$

Esempio: espressioni aritmetiche

La grammatica G:

$$G = (\{E, T, F\}, \{id, +, \times, '(', ')'\}, P, E)$$

ha l'insieme di regole di produzione P:
$E \to E + T \mid T$
$T \to T \times F \mid F$
$F \to (E) \mid id$
Il linguaggio di G:

$$L(G) = \{id, id + id, id \times id, (id + id) \times id, \dots\}$$

è l'insieme di espressioni aritmetiche generato. Dato che la grammatica G è ridotta e non circolare, il linguaggio $L(G)$ è infinito.

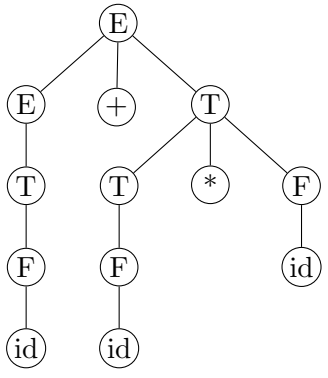
3.4 Albero di sintassi e derivazione

Il processo di derivazione può essere visualizzato come un albero, che descrive l'ordine in cui sono state derivate le parole.

La radice rappresenta l'assioma, mentre le foglie rappresentano i terminali. I nodi intermedi contengono solo nodi non terminali. Si può ricavare la parola leggendo le foglie da sinistra a destra.

Esempio: albero di derivazione delle espressioni aritmetiche

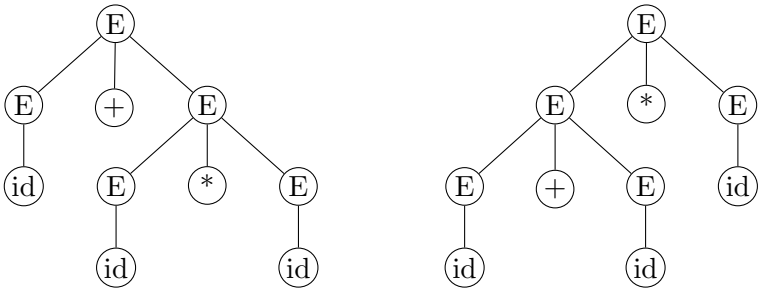
Visualizziamo l'albero di derivazione generato dal linguaggio riportato nel paragrafo precedente.



Per eliminare le ambiguità è possibile utilizzare delle parentesi, esplicitando la priorità tra operazioni

$$E \to E + E | E * E | id | (E)$$

Costruisco l'albero di derivazione per questa grammatica:



Notiamo che la stessa parola può essere generata in modi diversi (e quindi avere alberi di generazione differenti). Grammatiche di questo tipo sono dette *grammatiche ambigue*. Dal punto di vista di un compilatore questo tipo di grammatiche creano dei problemi perché non saprebbe quale strada scegliere.

Esistono linguaggi che si definiscono *sicuramente ambigui* perché per qualunque grammatica vado a scrivere risulta ambigua.

Per disambiguare la grammatica posso ampliarla inserendo altre regole:
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

Questa è una *grammatica stratificata*, in quanto ad ogni livello viene generata una sola operazione.

$$E \to E+T \to E+T*F \to E+T*id \to T+T*id \to F+T*id \to id+T*id \to id+F*id \to id+id*id$$

Questa grammatica non è ambigua.

3.5 Grammatiche di Chomsky e Greibach

Sulle grammatiche context free sono state definite varie forme normali (forme più vincolate che permettono, se le rispetto, di avere determinate proprietà). La prima forma normale che consideriamo è la **forma normale di Chomsky**, questa è applicabile a qualunque grammatica context free, ha solo due tipi di regole:

- da un non terminale si possono generare due non terminali:
  $A \to BC$ dove $A, B, C \in V$
- un non terminale può generare un simbolo dell'alfabeto o la stringa vuota ε:
  $A \to a$ dove $a \in \Sigma$

La caratteristica di queste regole è che l'albero di derivazione risulta essere sempre un *albero binario*, quindi ho sempre, quando non sono foglie, due figli.

Con la **forma normale di Greibach** si possono scrivere grammatiche context free con regole di un solo tipo:

- ogni non terminale genera un terminale:
  $A \to a\alpha$ dove $a \in \Sigma$ e $\alpha \in V^*$

La forma normale di Greibach dice: ogni volta che stai derivando qualcosa, o aggiungendo un livello all'albero, aggiungi un simbolo terminale alla produzione il più a sinistra possibile.

Un'altra caratteristica è che se devo generare una parola di lunghezza n faccio n derivazioni.

Un ultima cosa, importante per i compilatori e tutti gli automi su cui si basano i compilatori, è che la forma normale di Greibach evita di avere una ricorsione immediata sinistra.

Cosa non posso produrre con la forma normale di Greibach Nella forma normale di Chomsky posso produrre qualunque cosa, in quella di Greibach non è possibile produrre la stringa vuota.

3.6 Linguaggio lineare

Si intende per linguaggio lineare un linguaggio che nella parte destra delle regole di produzione ha al più un non terminale.

Le regole di produzione di un linguaggio context free sono: $V \times (\Sigma \times V)^*$, le regole per i linguaggi lineari sono:

$$A \to (\Sigma)^*(V \cup \{\varepsilon\})(\Sigma)^*$$

Nella parte destra della regola di produzione appare, al più, un non terminale.

3.6.1 Linguaggi lineari destre e sinistre

Nei linguaggi lineari dx o sx il carattere non terminale se dx va tutto a destra e viceversa per sx. Tutto ciò che è esprimibile con le lineari dx o sx è descrivibile con un linguaggio lineare.

Per ogni linguaggio lineare destra esiste una sinistra.

$$A \to \Sigma^*(V \cup \{\varepsilon\}) \quad dx$$

$$A \to (V \cup \{\varepsilon\})\Sigma^* \quad sx$$

Qualcosa che è lineare ma non dx o sx Nei linguaggi lineari dx o sx NON esiste un albero di crescita.

LINEARE:

$S \to aSb \mid ab$

LINEARE DX:

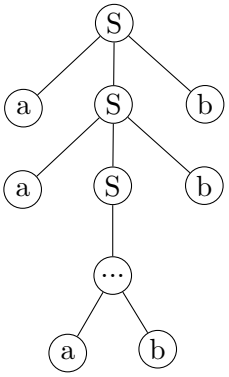
$S \to aS$

$S \to aB$

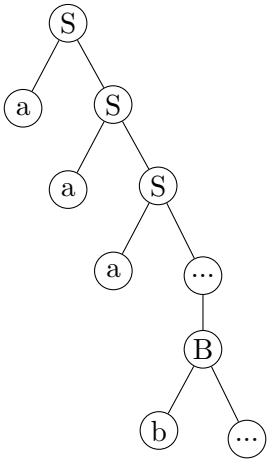
$B \to bB$

$B \to b$

LINEARE



LINEARE DX



Nel linguaggio lineare abbiamo un asse di crescita centrale, nel linguaggio lineare dx no, quindi non potrò mai bilanciare gli elementi perché metto gli elementi solo da una parte.

3.6.2 Linguaggi strettamente lineari dx o sx

Si aggiunge un ulteriore vincolo ai linguaggi lineari dx o sx, cioè: oltre ad avere un vincolo sulla presenza e posizione dei non terminali, aggiungo un vincolo sulla presenza dei terminali.

Strettamente lineare = se appare un terminale ne appare uno per volta. Al più un non terminale e, se c'è, un solo terminale.

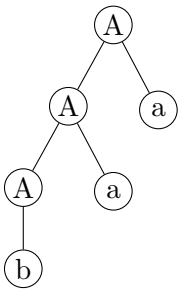
$$A \to (\Sigma \cup \{\varepsilon\})(V \cup \{\varepsilon\})$$

3.7 Conversione della ricorsione da sinistra a destra

Un'altra forma normale, detta *ricorsiva non-sinistra*, è caratterizzata dall'assenza di regole ricorsive sinistre o derivazioni (l-ricorsioni).

Trasformazione della ricorsione immediata sinistra

La grammatica che prenderemo in considerazione è: ba^*
$A \to Aa \mid b$



La b, regola di produzione che va a chiudere la ricorsione immediata sinistra, viene messa come primo terminale della parola.

Vorrei ottenere una grammatica che fa ottenere un albero speculare dove la b viene messa per prima.
$A \to bA'$ Come prima cosa scrivo la chiusura della ricorsione b, utilizzo un nuovo non terminale per la parte rimanente A';
$A' \to aA' \mid \varepsilon$
Oppure, rispettando la forma normale di Greibach:
$A \to b \mid bA'$
$A' \to aA' \mid a$
Consideriamo, in forma astratta, le regole di produzione che hanno una ricorsione immediata sinistra, tutte le alternative l-ricorsive per un non terminale A:

$$A \to A\beta_1 | A\beta_2 | \dots | A\beta_h \quad h \geq 1$$

dove nessuna stringa β_i è vuota, e siano le rimanenti alternative di A, che sono necessarie per terminare la ricorsione:

$$A \to \gamma_1 | \gamma_2 | \dots | \gamma_k \quad k \geq 1$$

Creiamo un nuovo terminale ausiliario A' e rimpiazziamo le regole precedenti con quelle mostrate in seguito:

$$A \to \gamma_1 A' | \gamma_2 A' | \dots | \gamma_k A' | \gamma_1 | \gamma_2 | \dots | \gamma_k$$
$$A' \to \beta_1 A' | \beta_2 A' | \dots | \beta_h A' | \beta_1 | \beta_2 | \dots | \beta_k$$

Ora, ogni derivazione che originalmente coinvolgeva passi l-ricorsivi, ad esempio

$$A \Rightarrow A\beta_2 \Rightarrow \beta_3\beta_2 A' \Rightarrow \gamma_1\beta_3\beta_2$$

è rimpiazzata dalla derivazione equivalente:

$$A \Rightarrow \gamma_1 A' \Rightarrow \gamma_1\beta_3 A' \Rightarrow \gamma_1\beta_3\beta_2$$

3.8 Confronto grammatica context free e lineare

L(Grammatica context free)	**L(Grammatica lineare)**
Definito da 1 quadrupla	Definito da 1 quadrupla

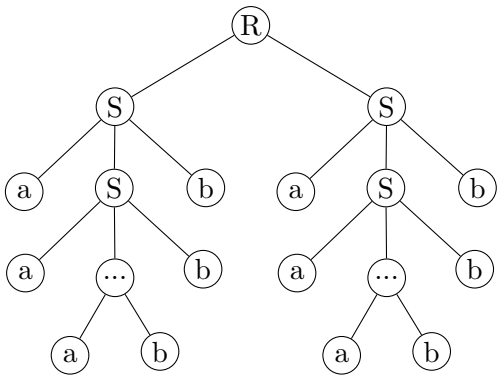
3.8.1 Grammatica lineare sx può generare tutto ciò che genera context free

Lineare $V \times (\Sigma)^*(V \cup \{\varepsilon\})(\Sigma)^*$
Context free $V \times (\Sigma \cup V)^*$
Tutto ciò che è contenuto nella grammatica context free è anche contenuto nella grammatica lineare.

3.8.2 Qualcosa context free ma non lineare

Lineare
$a^n b^n \qquad n > 0$
$S \to aSb \mid ab$

Context free
$a^n b^n a^m b^m \qquad n, m > 0$
$S \to aSb \mid ab$
$R \to SS$



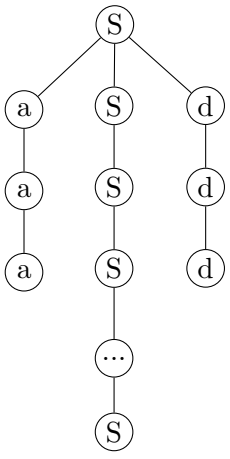
Sono presenti due assi di crescita nel linguaggio context free.

IMPORTANTE

IN UNA GRAMMATICA LINEARE PUÓ ESSERCI UN SOLO ASSE DI CRESCITA!
Linguaggi context free $\supset$ Linguaggi lineari

3.9 Confronto grammatica lineare e lineare dx/sx

$a^n d^n$
$S \to aSd$
$S \to \varepsilon$



Lineare destra
$S \to aS \mid D$
$D \to dD \mid \varepsilon$
Non ho un asse di crescita.

3.10 Confronto grammatica lineare destra e strettamente lineare destra

$S \to a_1 \cdots a_n$
$A \to a_1 \cdots a_n K$

Traduzione ↓
$S \to a_1 S_1 \cdots S_1 \to a_2 S_2 \cdots S_2 \to a_3 S_3$
$S \to a_1 S_1 \cdots S_1 \to a_2 S_2 \cdots S_{n-1} \to a_n K$

IMPORTANTE

La classe dei linguaggi lineari dx o sx sono EQUIVALENTI rispetto alla classe dei linguaggi strettamente lineari dx o sx.

L'equivalenza è possibile perché possiamo scomporre una regola lineare dx o sx in più regole strettamente lineari per soddisfare il linguaggio.

3.11 Confronto linguaggi lineari dx o sx e linguaggi basati su espressioni regolari

Vedremo come a partire da un linguaggio lineare dx o sx, si possa costruire l'espressione regolare equivalente. Se a partire da qualunque grammatica, lineare dx o sx, ho un modo da applicare per ottenere l'espressione regolare equivalente, non posso dire che sono equivalenti, ma posso dire che c'è un inclusione ($\subseteq$).
Partendo da una grammatica lineare dx abbiamo delle regole di questo tipo: $X \to \alpha_1 \mid \cdots \mid \alpha_k$, queste regole le posso trasformare in un equazione matematica: $X = \alpha_1 \cup \cdots \cup \alpha_k$, posso raccogliere per variabili i non terminali comuni: $X = (a_1 \cup a_n)X \cup (b_1 \cup \cdots \cup b_j)X_1 \cdots \cup (z_1 \cup \cdots \cup z_k)$, posso compattare tutto nel seguente modo:

$$X = \gamma X \cup \Omega$$

Dove γ rappresenta tutto quello che ho raccolto prima nella parentesi tonda e Ω rappresenta la parte restante, tutto quello che non è correlato al non terminale X. Questo porta alla soluzione:

$$X = \gamma^* \Omega$$

Questa equazione la uso per fare delle sostituzioni all'interno delle altre equazioni.

Esempio Consideriamo la seguente grammatica:
$S \to aB \mid aS$
$B \to dS \mid b$
Parto dalla grammatica e genero un sistema di due equazioni, il numero di equazioni sarà uguale al numero di non terminali da gestire (S, B).
$S = aB \cup aS$
$B = dS \cup b$
Posso trovare la soluzione nel primo e sostituirla nel secondo o viceversa. In tutti i modi arriverò alla soluzione, però mi accorgo che il secondo ha già la soluzione perché non c'è nessun B al suo interno.
$S = a(dS \cup b) \cup aS$
Per estrarre l'operatore di kleen devo sviluppare la parentesi tonda e poi raggruppare.
$S = ab \cup adS \cup aS$
Ora raggruppo.
$S = (ad \cup a)S \cup ab$
Ultimo step.
$S = (ad \cup a)^* ab$

3.11.1 Qualunque grammatica context free basata su un singolo terminale è esprimibile tramite un'espressione regolare

$S \to \alpha S \beta$
$S \to aSa \mid a$
In questa grammatica ho una regola ricorsiva auto inclusiva e genera $(aa)^*a$, $a^n a a^n$ con $n \geq 0$.

Fin tanto che per α e β ho dei non terminali la regola non esprime un espressione regolare. La cosa si complica se la ricorsione auto inclusiva è preceduta da un altro non terminale.
$S \to ASB \mid \varepsilon$
$A \to a$

$B \to \varepsilon$

Questo è un caso in cui ho una regola ricorsiva auto inclusiva ma alla fine il linguaggio è regolare.

$S \to ASB \mid b$

$A \to a$

$B \to \varepsilon$

Ora ho due terminali (a, b), in questo caso otterrò $A^n bB^n$, cioè $a^n b(\varepsilon)^n$ ma la stringa vuota è come se non esistesse quindi $a^n b \equiv a^* b$.

IMPORTANTE

Se in un linguaggio context free è presente un bilanciamento allora non esisterà un espressione regolare che lo descrive e di conseguenza non esisterà una grammatica dx o sx.

3.12 Forma normale di Von Dyck

La forma normale di Von Dyck permette, in una forma di linguaggio context free, di esprimere qualunque tipo di bilanciamento. Le regole sono:

$$S \to SS \mid a_i S c_i \mid \varepsilon$$

- SS ognuno potrebbe due non terminali differenti dove ognuno corrisponde a un diverso bilanciamento;
- a_i simbolo di apertura del bilanciamento;
- c_i simbolo di chiusura del bilanciamento;
- ε può essere sostituita dalla regola di chiusura delle ricorsione.

Un bilanciamento di questo tipo permette di produrre qualsiasi caso in un linguaggio di programmazione per le parentesi graffe.

```
{
    {
        {
        }
    }
}
```

Oppure

```
    {
    }
    {
    }
    {
    }
```

Per il secondo caso deriverei: $S \to SS \to SSS \to \{S\}SS \to \{\}SS \to \cdots \to \{\}\{\}\{\}$

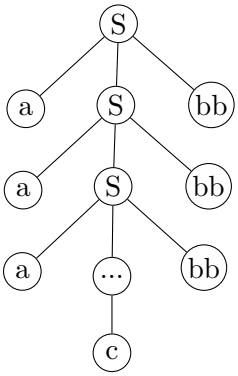
3.13 Grammatica dx/sx $\subseteq$ Linguaggi regolari

$S \to ASB$ Se c'è un bilanciamento non è un linguaggio regolare.
$A \to \varepsilon$
$B \to \varepsilon$

1.

2. Quanti sono i terminali? Qualunque grammatica context free monoterminale genera un linguaggio regolare.

$S \to aSbb \mid c$



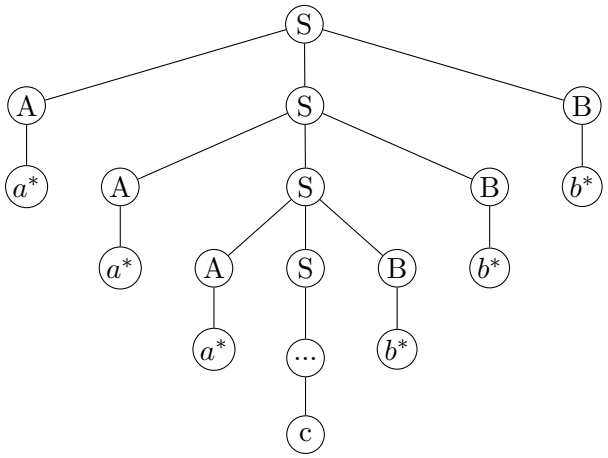
Esaustivo per 3 chiamate ricorsive.
$a^n c b^{2n}$ con $n \geq 0$

3.13.1 Ricorsione autoinclusiva

Genero qualcosa a sinistra, qualcosa a destra e qualcosa al centro.
$(a)^n c (bb)^n$

$S \to ASB \mid c$
$A \to aA \mid c$
$B \to bB \mid c$

$(a^*)^n c (b^*)^n$ con $n = 3$ avremo $a^*a^*a^*cb^*b^*b^*$



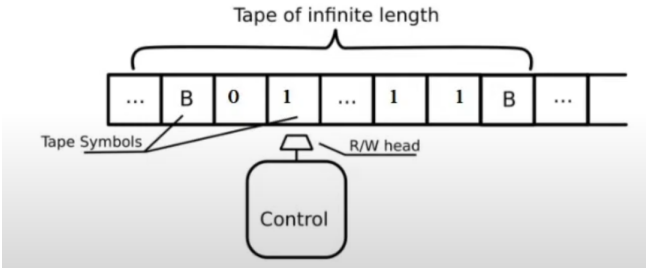
3.14 Gerarchia di Chomsky

Chomsky ha studiato e catalogato i linguaggi in base alla loro espressività, tipo di regole grammaticali che posso descrivere e in base alle tipologie di riconoscitori che servono per dire se una parola appartiene o no ad un determinato linguaggio.

Tipo	Forma delle produzioni	Linguaggio	Automa
3	$A \to \alpha X$ (lineare dx/sx)	Regolare	Automa a stati finiti det.
2	$A \to \gamma, \quad \gamma \in (\Sigma \cup V)^*$	Context-free	Automa a pila non det.
1	$\gamma \to \Omega, \quad \lvert\gamma\rvert \leq \lvert\Omega\rvert$	Context-sensitive	Macchina di Turing limitata
0	$\gamma \to \Omega, \quad \gamma, \Omega \in (\Sigma \cup V)^*$	Ricorsivamente enumerabile	Macchina di Turing

3.14.1 Macchina di Turing

Modello teorico che permette di calcolare tutto ciò che si può calcolare.



Caratteristiche:

1. Servono solo due simboli per computare (0, 1) e poi un simbolo particolare per indicare se la cella è vuota;
2. Il nastro ha lunghezza infinita.

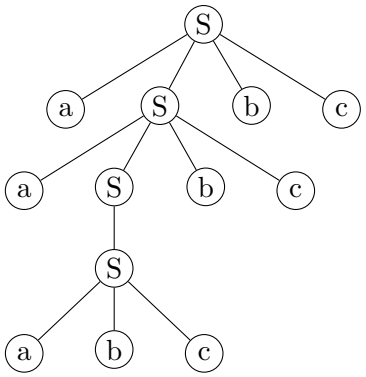
Com'è fatta?

1. Nastro di memoria;
2. Testina di lettura;
   - Legge;
   - Sovrascrive;
   - Muove il nastro da sinistra a destra;
3. Controller;
   - Contiene il programma.

3.15 Grammatica context sensitive

Voglio rappresentare $a^n b^n c^n$

Grammatica context free
$S \to aSbc$
$S \to abc$



Ottengo "$aaabcbcbc$" ma io volevo ottenere "$aaabbbccc$".

Rimuovendo "c" dalla ricorsione potrei ottenere "$aaabbb$" ma poi le "c" avranno un numero non correlato alle "a" e le "b".

Quindi non si può fare con un linguaggio context free.

Grammatica context sensitive
$S \to X \swarrow$
$X \to aXA \mid bXB$
$XA \to xA'$
$XB \to XB'$
$A'A \to AA'$
$A'B \to BA'$
$B'A \to AB'$
$B'B \to BB'$
$A' \swarrow \to a$
$B' \swarrow \to b$
$A'a \to aa$
$A'b \to ab$
$B'a \to ba$
$B'b \to bb$
$Xa \to ca$
$Xb \to cb$

4 Automi a stati finiti

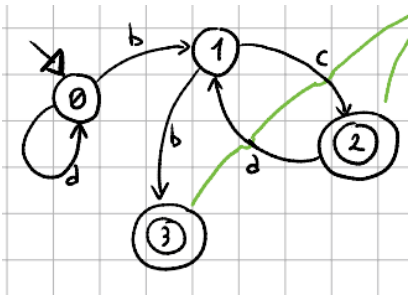
4.1 Automa riconoscitore

Per controllare se una stringa è valida per un determinato linguaggio, serve un algoritmo di riconoscimento, un tipo di algoritmo che produce una risposta positiva o negativa all'input fornito.

Gli automi sono rappresentati come grafi orientati. I nodi rappresentano gli stati; ogni arco è contrassegnato da un terminale e rappresenta la transizione causata dalla lettura di quel terminale.

Un automa è detto **deterministico** se consuma un simbolo alla volta.

Se non è presente un arco uguale al simbolo la risposta sarà NO. Quando consumo tutto l'input se l'ultimo stato in cui mi trovo è finale la risposta sarà SÍ altrimenti NO.



INPUT	STATO
bca	0
ca	1
a	2
	1

In questo caso essendo l'ultimo stato 1, non finale, la risposta sarà NO.

4.2 Definizione di un automa a stati finiti

Un automa a stati finiti è definito dalla 5-upla:

$$\langle Q, \Sigma, \sigma, q_0, F \rangle$$

con:

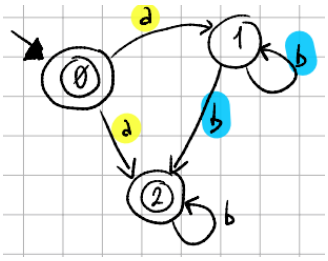
- Q: Insieme degli stati;
- Σ: alfabeto che descrive il linguaggio;
- q_0: stato iniziale;
- F: insieme degli stati finali;
- $\sigma(q_i, a) = q_s$: funzione di transizione (rappresentazione grafica dell'arco).

- $q_i, q_0 \in Q$
- $a \in \Sigma$

Se $\forall q_i, a \mid \sigma(q_i, a) \mid \leq 1$ allora è **deterministico**. In un automa non deterministico può assumere un qualsiasi numero reale.

4.3 Automi non deterministici senza ε mosse

Se ho più strade le devo fare tutte, dico NO se tutte lo dicono, SÍ se almeno una simulazione dice SÍ.



INPUT	STATO
ab	0
b	1
	1

NO

INPUT	STATO
ab	0
b	1
	2

SI

INPUT	STATO
ab	0
b	2
	2

SI

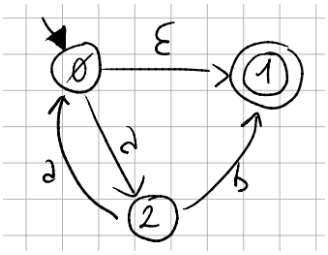
Sena ε mosse = sugli archi sono presenti solo label appartenenti all'alfabeto senza ε.

DETERMINISTICO $\sigma(q_i, a) = q_j$ $Q \times \Sigma \times Q$ $O(n)$

NON DETERMINISMO $\sigma(q_i, a) = \{q_1 \cdots q_j\}$ $Q \times \Sigma \times 2^Q$ $O(k^n)$ k=grado medio di non determinismo.

4.4 Automi non deterministici con ε mosse

L'arco ε viene attraversato senza consumare simboli dell'input. $Q \times \Sigma \cup \{\varepsilon\} \times 2^Q$



4.4.1 ε-chiusura

Una ε-chiusura è tutto quello che posso raggiungere tramite, e solo, attraversando archi ε.

DETERMINISMO a, q_i $\sigma(q_i, a)$

NON DETERMINISMO SENZA ε-MOSSE a, q_i $\sigma(q_i, a) = \{q_1 \cdots q_j\}$

NON DETERMINISMO CON ε-MOSSE q_i, a $\varepsilon - CH(\sigma(\varepsilon - CH(q_i), a))$

$\varepsilon - CH(q_{10}) = \{q_{10}, q_9, q_7\}$

$\sigma(q_{10}, a) = /$

$\sigma(q_9, a) = \{q_7, q_8\}$

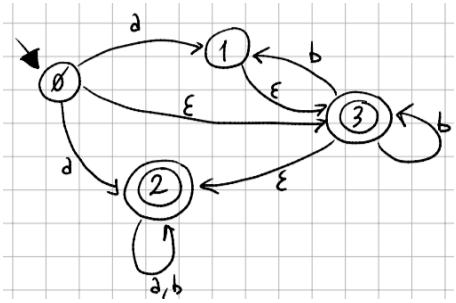
$\sigma(q_7, a) = \{q_7, q_{11}\}$

$\varepsilon - CH(q_7)$

$\varepsilon - CH(q_8)$

$\varepsilon - CH(q_{11})$

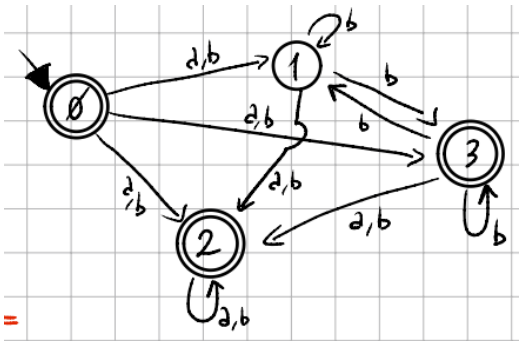
4.5 Trasformazione da ε-mosse a no ε-mosse



1. L'alfabeto non cambia;
2. Gli stati sono gli stessi;
3. Lo stato iniziale è lo stesso;
4. Calcolo la nuova funzione di transizione;
5. Gli stati finali rimangono tali, lo stato iniziale può diventare finale se all'inizio la stringa vuota da come risposta SÍ. **Se e solo se nell'ε -chiusura dello stato iniziale c'è uno stato finale**.

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

	0	**1**	**2**	**3**	
$\varepsilon - CH$	0,3,2	1,3,2	2	3,2	
$\sigma(\varepsilon - CH, a)$	1,2	2	2	2	
$\varepsilon - CH(\sigma(\varepsilon - CH, a))$	1,3,2	2	2	2	$\leftarrow$
$\sigma(\varepsilon - CH, b)$	1,2,3	1,2,3	2	1,2,3	
$\varepsilon - CH(\sigma(\varepsilon - CH, b))$	1,2,3	1,2,3	2	1,2,3	$\leftarrow$



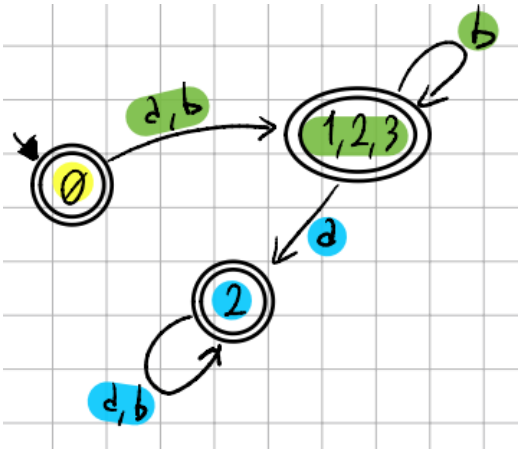
4.6 Eliminare il non determinismo

1. L'alfabeto rimane immutato;

2. Lo stato iniziale non cambia $Q \subseteq 2^Q$ Cardinalità $N \to 2^N$;
3. L'insieme degli stati cambia;
4. Gli stati finali sono quelli originali più tutti quelli che gli contengono.

	0^F	1	2^F	3^F	$0,1^F$	$0,2^F$	$0,3^F$	$1,2^F$	$1,3^F$	$2,3^F$	$0,1,2^F$	$0,1,3^F$	$1,2,3^F$	$0,1,2,3^F$
a	1,2,3	2	2	2	1,2,3	1,2,3	1,2,3	2	2	2	1,2,3	1,2,3	2	1,2,3
b	1,2,3	1,2,3	2	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3

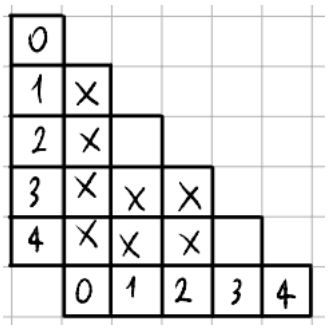
Tutto ciò che contiene 0 o 2 o 3 deve essere finale.



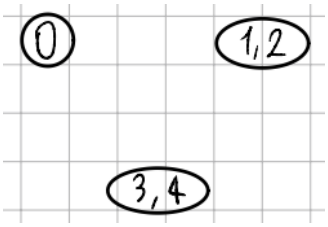
4.7 Rendere un automa minimo

Per rendere un automa minimo non elimino parti dell'automa inutili ma devo capire se 2 o più stati sono identici, cioè hanno lo stesso comportamento. Se sì posso fonderli tra loro.

1. Matrice senza diagonale;
2. Marcare le celle che sono diverse tra loro;



3. Creo l'automa con lo stato iniziale e le celle fuse.



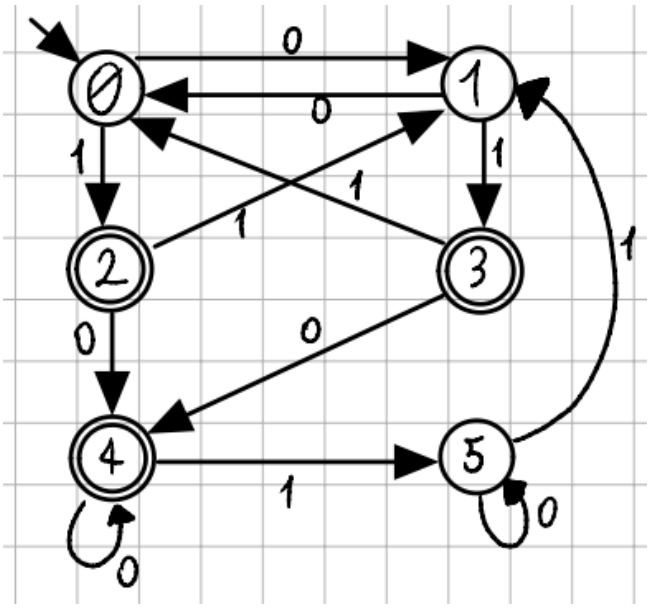
4.7.1 Calcolare se 2 stati sono identici

$p, q \in Q$
Criteri:

1. 1a) $p \in F \wedge q \in F$ *p e q sono stati finali*
   1b) $p \notin F \wedge q \notin F$ *p e q non sono stati finali*
2. $\forall x \in \Sigma^*$

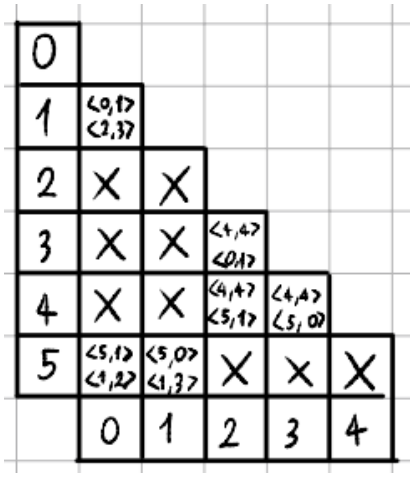
   2a) $\sigma^*(p, x) \in F \wedge \sigma^*(1, x) \in F$
   2b) $\sigma^*(p, x) \notin F \wedge \sigma^*(1, x) \notin F$

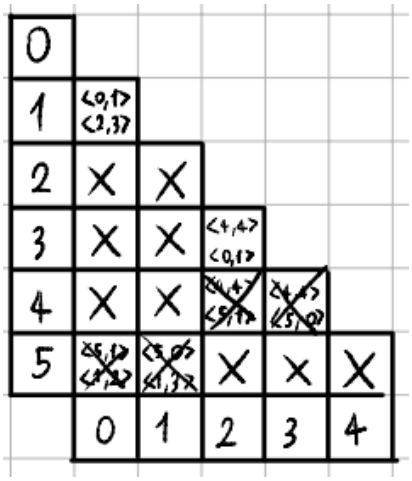
Esempio



0-1 per ogni simbolo scrivo delle coordinate

- Per 0 da dove arrivo per 0 da 0 e da 1;
- Se la cella è già marcata automaticamente la cella in questione viene marcata;
- Se vado sulla diagonale non verrà mai marcata;
- Se ho dei fallimenti i due stati non sono identici se in uno fallisco e nell'altro no. Per essere uguali devono fallire entrambi per il simbolo. Se sono diversi marco la cella.





2 e 3 erano stati finali.
Stato iniziale = quello che contiene lo stato iniziale originale.
Per gli archi controllo tutte le coppie stato simbolo nell'automa iniziale.

$\forall q \in Q \qquad \forall a \in \Sigma$
$\sigma(q, a) = S$
$S'([q], a) = [S]$

4.8 Da automa a stati finiti a grammatica lineare destra

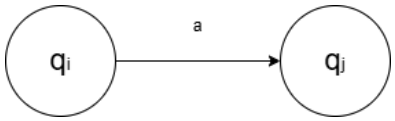
Automa a stati finiti $\langle Q, \Sigma, \sigma, q_0, F \rangle$

Grammatica lineare $\langle V, \Sigma, S, P \rangle$

L'alfabeto Σ è lo stesso.

L'idea è: in un automa ogni passo corrisponde ad un passo di riconoscimento, ogni volta che navigo un arco consumo un simbolo di input, per tradurlo in grammatica lineare trasformerò il riconoscimento in generazione.

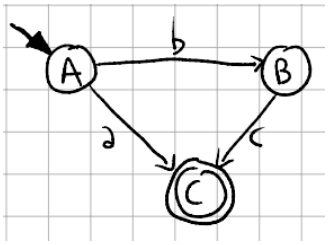
L'idea di base su cui si fonda la grammatica è:



$Q_i \to aQ_j$

Da questa idea base posso dedurre alcune cose:

- l'insieme dei non terminali è definito sulla base dell'insieme degli stati
  $V = Q$
- il nodo iniziale avrà un corrispondente non terminale, l'assioma
  $S = Q_0$
- per qualunque regola di produzione ottengo la regola equivalente
  $\forall \sigma(q_i, a) = q_j$ ottengo $Q_i \to aQ_j$



$V = \{A, B, C\} \quad S = A$

$A \to bB$
$A \to aC$
$B \to aA$
$B \to cC$
$C \to \varepsilon$
$\forall p \in P$
$P \to \varepsilon$

4.9 Da grammatica lineare Dx/Sx ad automa a stati finiti

Grammatica lineare $\langle V, \Sigma, S, P \rangle$

Automa a stati finiti $\langle Q, \Sigma, \sigma, q_0, F \rangle$

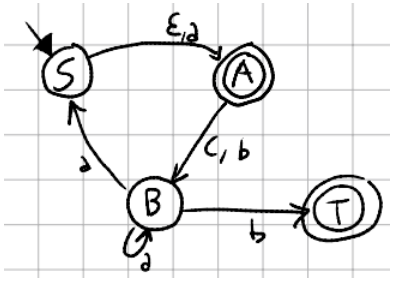
L'alfabeto Σ è lo stesso.

- Per ogni non terminale devo avere un corrispondente stato dell'automa
  $Q = V$

- $Q_0 = S$
- in una grammatica lineare dx posso trovarmi 4 tipi di regole:
  - $A \to aB$ scriverò la funzione di transizione $\sigma(A, a) = B$
  - $A \to B$ scriverò la funzione di transizione $\sigma(A, \varepsilon) = B$, ε-mossa
  - $\forall A \to a$ devo creare un nuovo stato T finale $\sigma(A, a) = T$, dove $T \in F$
  - $\forall A \mid A \to \varepsilon$ allora $A \in F$

Esempio:
$S \to A \mid aA$
$A \to cB \mid bB \mid \varepsilon$
$B \to aS \mid aB \mid b$

$\forall p \in V$ c'è $q \in Q$
$q_0 = S$

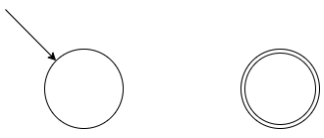


1. $x \to \alpha Y \qquad \alpha \in (\Sigma \cup \{\varepsilon\} \implies \sigma(x, \alpha) = Y$
2. $x \to a \qquad a \in \Sigma \implies \sigma(x, \alpha) = t \qquad t \in F$
3. $x \to \varepsilon \implies x \in T$

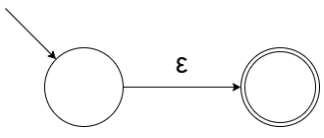
4.10 Da espressione regolare ad automa a stati finiti

Algoritmo di **Thomson**, solo per grammatiche lineari destre.
Casi base:

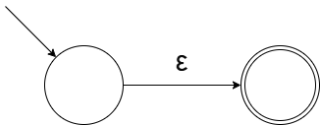
1. L'insieme vuoto;



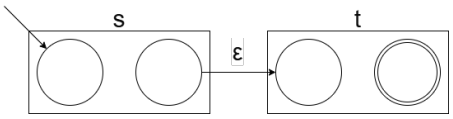
2. La stringa vuota;



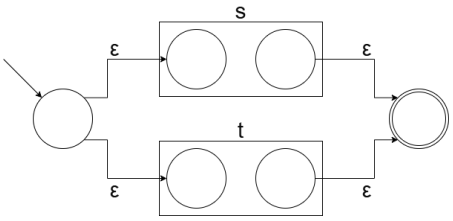
3. Una parola di un singolo simbolo;



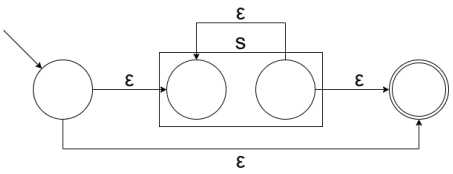
4. (a) $s \cdot t$ e.r $\Longrightarrow$



(b) $s \cup t$ e.r $\Longrightarrow$

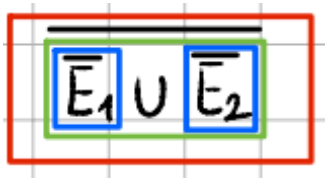


5. S^* e.r



Dati 2 linguaggi regolari, se faccio l'intersezione dei due ($E_1 \cap E_2$) ottengo ancora un espressione regolare?
De Morgan

$$\overline{\overline{E_1} \cap \overline{E_2}} = \overline{\overline{E_1} \cap \overline{E_2}}$$



Blu = Espressione regolare.
Verde = L'unione di 2 espressioni regolari è una espressione regolare.
Rosso = Il negato di una espressione regolare è una espressione regolare.

5 Automi a pila e parsing

5.1 Automi a pila

Gli *automi a pila* sono automi a stati finiti che utilizzano una *pila* (stack) come memoria aggiuntiva.

Uso la pila per contare, quando incontro un elemento, es. a, faccio una push trovo b e faccio pop se alla fine la pila è vuota avrò lo stesso numero di "a" e di "b", $a^n b^n$

Un automa a pila è definito dalla 7-upla

$$\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$$

con:

- Q: insieme degli stati
- Σ: alfabeto che descrive il linguaggio
- Γ: alfabeto della pila

- σ: funzione di transizione
- q_0: stato iniziale
- Z_0: fondo della pila
- F: stato (o stati) finale

5.1.1 Funzione di transizione σ

Non è più una tripla ma una quintupla (3 in, 2 out).

$$\sigma(Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) = 2^{Q \times \Gamma^*}$$

L'input è una tripla, denotata come:

$$(q, a, A) \to (x, XX)$$

con:

- q: stato corrente
- a: il simbolo della stringa da leggere
- A: il contenuto dello stack

Il simbolo di fine stringa è $\swarrow$.

5.1.2 Tipi di accettazione

Negli automi a pila ci sono due tipi di accettazione: l'*accettazione per stato finale*, quando è stato consumato tutto l'input e si giunge ad uno stato finale, e l'*accettazione per pila vuota*, quando è stato consumato tutto l'input e la pila è vuota (anche senza Z_0).

Essendo l'automa non deterministico, bisogna fare tutte le computazioni possibili (quindi esplorare tutte le possibilità).

5.1.3 Esempio di accettazione per stato finale

Processiamo una stringa nella forma ca^nb^n con $n \geq 1$, ad esempio $caabb\swarrow$.

$$\Gamma = \{Z_0, X\}$$

Z_0 è sempre presente, X è il simbolo che gestisce il bilanciamento. Consumando c, si passa dallo stato q_0 allo stato q_1.

$$(q_0, c, Z_0) \to (q_1, Z_0)$$

Ora non sarà più possibile incontrare c. Consumiamo a:

$$(q_1, a, Z_0) \to (q_1, Z_0X)$$

X serve a contare le a. La funzione (q_1, Z_0X) rimane in q_1; la testa della pila contiene X, quindi la funzione (q_1, a, Z_0) non può scattare. Bisogna definire una nuova:

$$(q_1, a, X) \to (q_1, XX)$$

Se la parola è corretta, prima o poi si incontrerà una b. Passiamo allo stato q_2 per non incontrare più a.

$$(q_1, b, X) \to (q_2, \varepsilon)$$

Non può esserci Z_0, altrimenti sarebbe come se non avessimo mai incontrato nessuna a. Il passaggio a q_2 è obbligato.

$$(q_2, b, X) \to (q_2, \varepsilon)$$

$$(q_2, \swarrow, X) \to (q_3, Z_0)$$

Incontrerò il fine stringa quando aavrò rimosso tutti gli X dalla pila. Lo stato finale conterrà solo q_3.

Input	Pila	Stato	Commenti
$caabb \swarrow$	Z_0	q_0	devo consumare c e Z_0
$aabb \swarrow$	Z_0	q_1	devo consumare a
$abb \swarrow$	Z_0X	q_1	devo trovare tripla (q_1, a, X)
$bb \swarrow$	Z_0XX	q_1	ogni volta che incontro una a, metto X sulla pila
$b \swarrow$	Z_0X	q_2	consumo la testa della pila, non scrivo nulla
$\swarrow$	Z_0	q_2	
	Z_0	q_3	la parola appartiene al linguaggio

5.1.4 Regole di produzione

Una *grammatica context free* genera da un non terminale una sequenza di terminali e non terminali, combinati in qualunque modo; è una quadrupla nella forma

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

È possibile usare l'automa a pila per simulare la fase di generazione: quando trovo un non terminale, posso sostituirlo con un terminale o un non terminale.

La costruzione della funzione di transizione viene guidata dalle regole di produzione. Il funzionamento dell'automa a pila è il seguente: controllo l'elemento in cima alla pila, individuo la regola di produzione corrispondente e la applico.

Esistono 4 categorie di regole di generazione: regola di *inizializzazione*, regola di *terminazione*, regole *derivate da* P e regole *derivate da* Σ. Per qualunque tripla, si può applicare più di una regola.

Inizializzazione Questa regola permette di far partire la generazione, corrisponde a mettere sulla pila l'assioma S.

$$(q_0, \varepsilon, Z_0) \to (q_0, \swarrow S)$$

Implico il trovarmi in q_0 e dover transizionare in q_0. Non consumo nulla, ma modifico il contenuto della pila. Accettando per pila vuota, non bisogna includere Z_0.

Terminazione Questa regola permette di terminare la generazione; l'ultimo simbolo in input è quello di fine stringa ($\swarrow$).

$$(q_0, \swarrow, \swarrow) \to (q_0, \varepsilon)$$

La generazione termina quando l'automa incontra il simbolo di fine stringa. Non viene scritto nulla sulla pila, ma si rimuove $\swarrow$, terminando la generazione.

Regole per Σ Esiste una regola per ogni simbolo dell'alfabeto ($\forall a \in \Sigma$).

$$(q_0, a, a) \to (q_0, \varepsilon)$$

Il simbolo in cima alla pila viene consumato. Esistono due tipi di regole di produzione per a, quelle che *iniziano con un terminale*

$$(q_0, a, A) \to (q_0, \beta^R) \quad per \quad A \to a\beta$$

e quelle che *iniziano con un non terminale*

$$(q_0, \varepsilon, A) \to (q_0, \beta^R X) \quad per \quad A \to X\beta$$

5.1.5 Esempio di accettazione per pila vuota

Creiamo un automa a stati finiti non deterministico che accetta per *pila vuota*:

- $Q = \{q_0\}$: perchè si può gestire il tutto con un solo stato (dato il non determinismo) e l'insieme degli stati finali è vuoto.
- $\Sigma = \Sigma$: l'alfabeto è quello del linguaggio

- $\Gamma = \{Z_0, \dots\}$: conterrà sicuramente il simbolo di fine pila, più tutti i simboli scrivibili sulla pila
- $F = \{\emptyset\}$: l'insieme degli stati finali è vuoto

L'alfabeto della pila è definito come

$$\{Z_0\} \cup \Sigma \cup V$$

ovvero l'unione del simbolo di fine pila e gli insiemi dei simboli terminali e non terminali.

Regole di produzione:

$$S \to aBA \qquad S \to bcS \qquad B \to Ba \qquad B \to A \qquad A \to ac \qquad A \to AA$$

La funzione di transizione è composta da 11 regole. Le seguenti regole di inizializzazione e terminazione

$$(q_0, \varepsilon, Z_0) \to (q_0, \swarrow S) \qquad\qquad (q_0, \swarrow, \swarrow) \to (q_0, \varepsilon)$$

sono comuni a tutti i linguaggi.

Le regole

$$(q_0, a, a) \to (q_0, \varepsilon) \qquad (q_0, b, b) \to (q_0, \varepsilon) \qquad (q_0, c, c) \to (q_0, \varepsilon)$$

non scrivono nulla sulla pila.

Infine

$$\begin{array}{lll} (q_0, a, S) \to (q_0, AB) & (q_0, b, S) \to (q_0, Sc) & (q_0, \varepsilon, B) \to (q_0, aB) \\ (q_0, \varepsilon, B) \to (q_0, A) & (q_0, a, A) \to (q_0, c) & (q_0, \varepsilon, A) \to (q_0, AA) \end{array}$$

Generiamo la stringa *aacac* sequendo le regole di produzione ed esaminiamola.

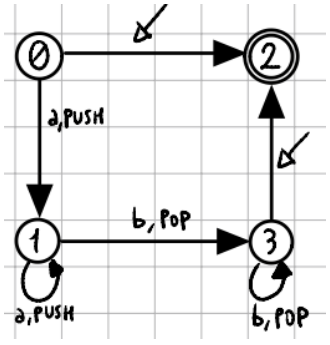
$$S \to aBA \to aAA \to aacA \to aacac$$

Input	Pila	Stato	Regola di produzione
$aacac\swarrow$	Z_0	q_0	$(q_0, \varepsilon, Z_0) \to (q_0, \swarrow S)$
$aacac\swarrow$	$\swarrow S$	q_0	$(q_0, a, S) \to (q_0, AB)$
$acac\swarrow$	$\swarrow AB$	q_0	$(q_0, \varepsilon, B) \to (q_0, A)$
$acac\swarrow$	$\swarrow AA$	q_0	$(q_0, a, A) \to (q_0, c)$
$cac\swarrow$	$\swarrow Ac$	q_0	$(q_0, c, c) \to (q_0, \varepsilon)$
$ac\swarrow$	$\swarrow A$	q_0	$(q_0, a, A) \to (q_0, c)$
$c\swarrow$	$\swarrow c$	q_0	$(q_0, c, c) \to (q_0, \varepsilon)$
$\swarrow$	$\swarrow$	q_0	$(q_0, \swarrow, \swarrow) \to (q_0, \varepsilon)$

5.1.6 Come funziona un automa a pila

$a^n b^{2n}$ $n \geq 0$ Ipotizziamo di avere sempre il simbolo di fine stringa $\swarrow$.

Funzione di transizione per automa a pila con accettazione per stato finale



$(q_0, \swarrow, Z_0) \implies (q_2, \swarrow)$

$(q_0, a, Z_0) \implies (q_1, Z_0XX)$

$(q_1, a, x) \implies (q_1, XXX)$

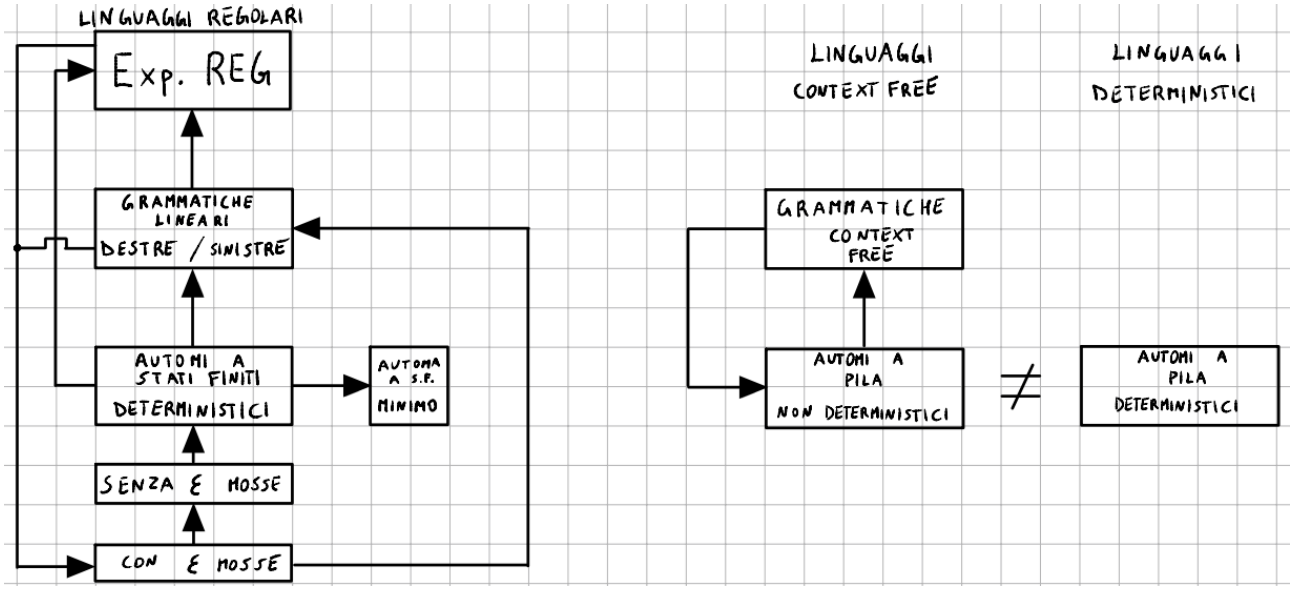
$(q_1, b, X) \implies (q_3, \varepsilon)$

$(q_3, b, X) \implies (q_3, \varepsilon)$

$(q_3, \swarrow, Z_0) \implies (q_2, Z_0)$

PILA	INPUT	STATO
Z_0	aabbbb$\swarrow$	q0
Z_0XX	abbbb$\swarrow$	q_1
Z_0XXXX	bbbb$\swarrow$	q_1
Z_0XXX	bbb$\swarrow$	q_3
Z_0XX	bb$\swarrow$	q_3
Z_0X	b$\swarrow$	q_3
Z_0	$\swarrow$	q_3
Z_0		q_2

5.2 Confronto Linguaggi Regolari, Linguaggi Context Free e Linguaggi Deterministici



5.3 Parsing

L'*albero di derivazione* è creato durante la parsificazione, questo fornisce l'interpretazione della parola in input che si vuole verificare.

5.3.1 Tipi di parser

Parser top-down Si parte dalla root e si costruisce fino ad arrivare alle foglie che rappresentano i terminali.

Parser bottom-up Si parte dalle foglie e tenta di ricostruire l'albero fino ad arrivare alla root dell'albero che contiene l'assioma.

Parser di questo tipo sono automi a pila.

5.3.2 Parser di tipo $LR(0)$

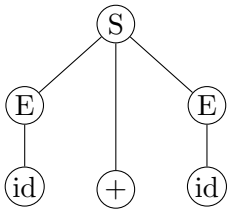
Vediamo un parser di tipo $LR(0)$.

Con 0 intendiamo che, oltre a consumare un simbolo in input, *legge 0 altri simboli.*

Con L intendiamo *left*: il parser parte da sinistra con la lettura.

Con R intendiamo *rightmost*: il parser cerca la regola della grammatica da utilizzare partendo da quella più a destra.

Si inseriscono nodi nell'albero ogni volta che si effettua una riduzione. Ad esempio, date le seguenti regole di produzione:
$E \to id$
$S \to E + E$
si ottiene l'albero



$LR(0)$ è un'*automa a pila deterministico*: in ogni momento della parsificazione è possibile compiere una sola azione (o nessuna). Il suo compito è accettare o rifiutare una stringa in input. Sono inoltre possibili quattro operazioni:

- *SHIFT*: leggo input e lo trascrivo sulla pila
- *REDUCE*: operazione legata ad una regola grammaticale; consuma simboli dalla pila e li sostituisce
- *ACCEPT*
- *FAIL*

L'operazione di REDUCE non modifica la pila, fa una serie di pop e poi fa una push.

Finora, gli stati sono stati identificati per label. In $LR(0)$ gli stati sono etichettati con "SHIFT" o "REDUCE" e contengono informazioni utili a determinare il tipo di operazione da svolgere.

Un parser $LR(0)$ non gestisce tutte le grammatiche context free, ma è possibile costruire un parser a partire da una di queste.

Durante la parsificazione di possono verificare due problemi:

- il comportamento non è deterministico: alcuni stati hanno due o più comandi
- si possono avere più operazioni di reduce, ognuna legata ad una regola diversa (qual è quella corretta)

Inoltre, *non* è possibile fare contemporaneamente operazioni di SHIFT e REDUCE oppure due operazioni di REDUCE in parallelo.

Un automa a pila deterministico ha all'interno dei suoi stati dei candidati legati alla regola di produzione. $A \to a^{\beta}$

5.4 Come costruire un parser

1. Calcolare le chiusure;
2. Etichettare gli stati (comandi=
   - SHIFT = marker non alla fine;
   - REDUCE = marker alla fine;
3. Dire se è un parser LR(0) = se trovo una situazione con più di un comando contemporaneamente non è un parser LR(0) perché non deterministico.
   - Shift/Reduce: non si riesce a decidere se devo fare un'operazione di shift o reduce;
   - Reduce/Reduce: una parte riducibile ha due alternative differenti.

5.5 Parser LR(k) k=1

Posso vedere un simbolo a destra senza consumarlo. Utile quando ho dei conflitti per decidere quale operazione fare nei momenti non deterministici.

Non necessariamente tutti i conflitti possono essere risolti.

IMPORTANTE

Se un parser è LR(0) è automaticamente SLR(1)!

5.6 Parser SLR(1)

S = simple, il più semplice parser LR(1).

Dopo aver costruito un parser LR(0) ed aver trovato tutti i problemi passo ad SLR(1) dove non cambia la struttura del parser.

Dove c'è una reduce devo aggiungere un'informazione (insieme dei **follow**).

First(x) primo simbolo che appare in una sequenza, x identifica una sequenza di terminali o non terminali.

1. $x \in Z \qquad First(x) = \{x\} \qquad$ se x è un terminale;
2. $x \in Z \quad e \downarrow \qquad$ se x è un NON terminale;
   - $x \to a\alpha \qquad a \in First(x)$
   - $x \to \varepsilon \qquad \varepsilon \in First(x)$
3. $x \to YZ \qquad se \downarrow$
   - $\varepsilon \notin First(Y) \qquad First(x) = First(Y)$
   - $\varepsilon \in First(Y) \qquad First(X) = (First(Y) - \{\varepsilon\}) \cup First(Z)$

Follow dato un non terminale, quello che può succedere immediatamente dopo dei terminali.
$Follow(M) : M \in V$ ci servono tutte le produzioni che riguardano M dove M appare nella parte destra. $N \to \alpha M \beta$

- ADD $First(\beta) - \{\varepsilon\}$
- $\varepsilon \in First(\beta)(o \quad \beta = \varepsilon) \qquad ADD \quad Follow(N)$