I linguaggi regolari sono chiusi per intersezione: Dati due linguaggi L_1, L_2 regolari, $L_1 \cap L_2$ è regolare.

DIMOSTRAZIONE

Se L_1 ed L_2 sono linguaggi regolari, $\exists$ DFA M_1, M_2 t.c. $L(M_1) = L_1$ & $L(M_2) = L_2$.

Supponiamo che $M_1 = (S_1, A_1, d_1 := move_{d_1}, s_0, F)$ e $M_2 = (S_2, A_2, d_2 := move_{d_2}, s_{0_2}, F_2)$.

CASO 1: $A_1 \cap A_2 = \phi$, cioè parlano di simboli completamente diversi.

Allora sia $\bar{M} = (\{s_{0_1}\}, A_1, \perp, s_{0_1}, \phi)$ è t.c. $L(\bar{M}) = L_1 \cap L_2 = \phi$.

↑ uno a caso dei due

CASO 2: $A_1 \cap A_2 \neq \phi$.

Allora sia $A = A_1 \cup A_2$ e M_{1_A} e M_{2_A} i DFA sull'alfabeto con funzione di transizione totale ed equivalenti rispettivamente a M_1 e M_2.

Sia $M = (S, A, \delta, s_0, F)$ in cui S contiene stati che sono ottenuti come coppie (s_i, s_j) per $s_i \in S_1^A$ e $s_j \in S_2^A$ (prodotto cartesiano).

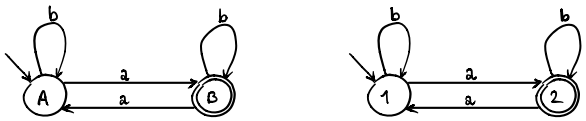
δ è t.c. $\delta((s_i, s_j), a) = (s_i', s_j') \Leftrightarrow S_1^A(s_i, a) = s_i' \in S_2^A(s_j, a) = s_0'$. s_0 è lo stato (s_0', s_0^2).

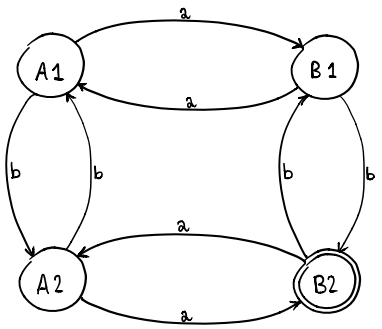
$F \{(s_i, s_j) \mid s_i \in F_1 \text{ e } s_j \in F_2\}$

Allora $w \in L(M) \Leftarrow$ dallo stato iniziale di M_{1_A} $\exists$ un cammino a $(s_i, s_j) \in F \Longleftrightarrow$

dallo stato iniziale di M_{2_A} $\exists$ un cammino a $s_j \Longleftrightarrow$

$w \in L_1 \cap L_2$

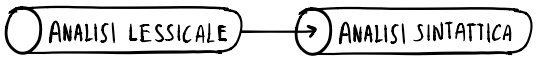




STOP AUTOMI,

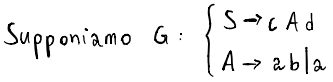
ORA ANALISI SINTATTICA

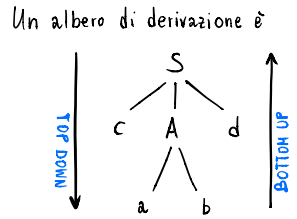
L'analisi lessicale, che si basa sugli automi a stati finiti, riguarda appunto il lessico, le parole. In nessun modo riguarda la *struttura* delle frasi, che invece è la preoccupazione in assoluto principale dell'analisi sintattica, anche se, come vedremo, all'analisi sintattica ci si può agganciare una serie di controlli che, alla fine, riescono a farci capire...



L'analisi sintattica si occupa, in primo luogo, di stabilire se una certa stringa (l'output dell'analisi lessicale) sia in effetti derivabile dalla grammatica del linguaggio.

Vi sono due grosse categorie di analizzatori sintattici: si parla di analisi **top-down** o analisi **bottom-up**. La differenza sta nella maniera in cui si costruisce l'albero di derivazione.

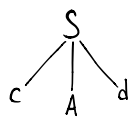




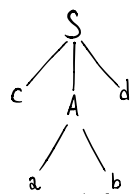
Top-down: costruisce l'albero dalla radice alla frontiera, ed è per questo più facile da capire.
Bottom-up: costruisce l'albero dalla frontiera alla radice (black magic?); la gran parte delle grammatiche dei linguaggi di programmazione sono analizzabili in bottom-up, e pertanto la gran parte degli analizzatori usano questa categoria.

Curiosità — Il top-down è il primo che è stato pensato ma è stato poi abbandonato, anche se recentemente è rientrato nelle grazie di alcuni, perché si può usare per fare dei controlli che non possono essere fatti in fase di analisi lessicale. Questo, ad esempio, nei linguaggi di programmazione che utilizzano l'indentazione per strutturare in blocchi: il 'tab' è già struttura, e quindi a livello di analisi lessicale non si vede.

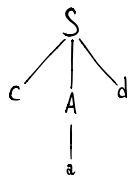
Simulazione
"`cad`" ∈ L(G)? Devo poter ottenere un albero di derivazione per quella stringa.
La mia stringa inizia con la '`c`': guardo tutte le produzioni della S che iniziano con la '`c`'. In questo caso sono fortunato, perché ho solo una produzione che espande la S, e quindi certamente devo usare quella.



Ho matchato la '`c`'. Adesso devo fare qualcosa che generi '`a`', ossia che porti '`a`' sulla frontiera dell'albero. Il prossimo simbolo che vedo, sulla mia frontiera in costruzione, è il non-terminale A. Qui sono un po' meno fortunato rispetto al caso precedente, perché ho ben due produzioni che iniziano con la '`a`'. Che fare? Diciamo che provo la prima.



Ho matchato la '`a`'; adesso mi aspetto di incontrare una '`d`', ma così non è, in quanto in contro una '`b`'. Provo ad andare indietro (backtrack), controllando se posso ottenere di meglio.

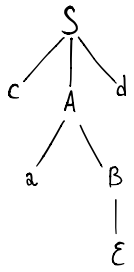


Con questo secondo tentativo, matcho la '`a`' nuovamente, dopodiché mi aspetto di matchare la '`d`', il che accade. Mi accorgo che non ho più nessun non-terminale da espandere; l'albero è un albero di derivazione ben formato della stringa "`cad`" e quindi si conclude che tale stringa è derivabile.

Questo algoritmo è ricorsivo e usa il **backtrack** in caso di fallimento. Le tecniche di top-down utilizzate oggi sono **ANTLR**: invece di fare ricorso al backtracking, si fa uso di un top-down **predittivo**, in cui si modifica un po' la grammatica per far tornare bene le cose.

Una qualche modifica della grammatica precedente che semplifica la vita evitando di fare backtrack è la seguente:

$$\begin{cases} S \to c\,A\,d \\ A \to a\,B \\ B \to b \mid \varepsilon \end{cases}$$



Dovremo capire quali sono le caratteristiche delle grammatiche che non ci fanno fare questo top-down predittivo, e quindi attrezzarci per trasformarle in grammatiche che lo consentano.

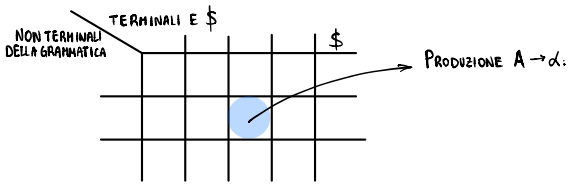
Analizzatore sintattico

Strutture dati: - input buffer - pila - parsing table

Input buffer: parola + terminatore: w $

Pila: inizializzata così: [$] la testa della pila contiene la derivazione corrente. Serve ad immagazzinare la porzione di frontiera che ancora deve essere matchata con la parte dell'input che non abbiamo ancora matchato.

Parsing table: nel caso del top-down predittivo, anche conosciuto come LL(1), indica come espandere un certo non-terminale quando nell'input buffer vediamo un certo simbolo.



ESEMPIO:

$$\begin{cases} S \to c\,A\,d \\ A \to a\,B \\ B \to b \mid \varepsilon \end{cases}$$

NON TERMINALI	a	b	c	d	$
S	ERROR	...	S→cAd	...	...
A	A→aB	...	...	...	...
B	...	B→b	...	B→ε	...

Per compilare la tabella abbiamo bisogno dei FIRST e dei FOLLOW, che vedremo nella lezione successiva:

	First	Follow
S	c	$
A	a	d
B	b, ε	d

Algoritmo:

INPUT: w, parsing table M per G.

OUTPUT: derivazione leftmost di w se $w \in L(G)$, altrimenti error( ).

INIZIALIZZAZIONE: input buffer ← w$, pila [$ S]

```
1   let b il primo simbolo di w
2   let X il top della pila
3   while (X≠$) {
4       if (X=b) {
5           pop(X);
6           b= simbolo successivo nell'input buffer
7       }
8       else if (X è un terminale)
9           error();
10      else if (M[X,b] è error())
11          error();
12      else if (M[X,b] = X→Y1...Yk) {      % per convenzione, X e Y grandi possono rappresentare sia non-terminali che terminali.
13          output(X→Y1...Yk);
14          pop(X);
15          push(Yk...Y1);
16      }
17      X= top della pila
18  }
```

ESEMPIO: Riconoscimento di "cad" INPUT BUFFER: cad$, PILA: $S