

01. Data la grammatica, determina il linguaggio (13 sett)	3
02. Dato il linguaggio, determina la grammatica (14 sett)	5
03. Pumping lemma (18 sett)	8
04. Proprietà insiemistiche dei linguaggi liberi (20 sett)	10
05. Dimostrazione intuitiva del pumping lemma (20 sett)	11
06. Grammatiche per i numeri naturali (21 sett)	13
07. Introduzione all'analisi lessicale (25 sett)	14
08. NFA (27 sett)	15
09. Simulazione di un NFA (28 sett)	18
10. DFA (2 ott)	20
11. DFA minimo (3 ott)	23
12. Partition refinement (intuitivo) (4 ott)	25
13. Minimizzazione DFA (5 ott)	27
14. Esercizi minimizzazione (9 ott)	29
15. Pumping lemma per linguaggi regolari ed Epsilon chiusura (18 ott)	30
16. Parsing (19 ott)	32
17. First e follow (23 ott)	35
18. Follow (Libro di Abeni) (23 ott)	37
19. Esercizi su follow (25 ott)	39
20. Grammatiche LL(1) (26 ott)	41
21. Eliminare la ricorsione a sinistra (8 nov)	43
22. Curare la ricorsione (9 nov)	45
23. Bottom-up parsing - introduzione (14 nov)	48
24. Parsing shift-reduce (15 nov)	51
25. Costruzione automa caratteristico per parsing LR(1) canonico (16 nov)	53
26. Chiusura (21 nov)	55
27. Esercizi tabelle di parsing canonical LR(1) (22 nov)	59
28. Conflitti nella tabella e loro risoluzione (23 nov)	62
29. LALR (28 nov)	66
30. Merged automata (29 nov)	70

31. Sistema di equazioni (30 nov)	77
32. Syntax-directed definitions e attributi (6 dic)	80
33. SDD S-attribuite e T-attribuite (7 dic)	84
34. Esercizi SDD (12 dic)	87
35. Curiosità varie (13 dic)	88
36. Generazione del codice intermedio (14 dic)	89

Programma: sequenze di stringhe. Stringa = sequenze di caratteri.

Abbiamo dato una definizione formale di **grammatica generativa**. Vediamone una definizione informale:

Una grammatica generativa è un insieme di regole che "specificano" o "generano" in modo ricorsivo le formule ben formate di un linguaggio. Una formula ben formata è una stringa di simboli che, intuitivamente, rappresenta un'espressione sintatticamente corretta, e viene definita mediante le regole della grammatica.

Formalmente, una grammatica è una tupla quadrupla:

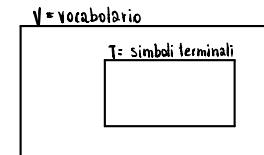
$$G = (V, T, S, P)$$

V è il vocabolario, che comprende simboli non-terminali e simboli terminali.

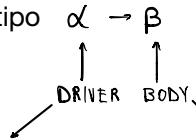
T è l'insieme dei simboli terminali, sottoinsieme di V.

S è il simbolo iniziale, scelto tra i simboli non-terminali.

P è l'insieme delle **produzioni**, che indicano una possibile forma di un costrutto.



La forma delle produzioni è del tipo $\alpha \rightarrow \beta$



$\alpha \in V^+$, cioè è una stringa costituita da uno o più simboli di V,
e' vincolo che almeno un simbolo è $V \setminus T$.

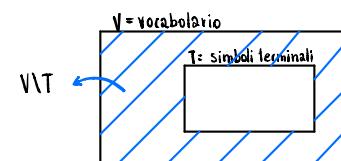
$\beta \in V^*$, ossia è una stringa costituita da 0 (stringa vuota $\beta = \epsilon$) o più elementi.

Notazione:

- V^+ : il + indica la ripetizione di una o più volte dell'oggetto alla base, la V. Si forma quindi una stringa lunga 1, 2, 3... n.
- V^* : la * indica la ripetizione di zero o più volte dell'oggetto alla base, la V. Si forma quindi una stringa lunga 0, 1, 2, 3... n.
Quando è lunga 0, β è ϵ .

Un esempio di grammatica: $G = (\{S, a, b\}, \{a, b\}, S, \{S \rightarrow a, S \rightarrow b\})$

vocabolario di 3 simboli: nelle produzioni
che vado a scrivere posso comporre solo
questi 3 simboli.



Produzione: **stringa \rightarrow stringa**.

↑
Pesca simboli e V; potrebbe essere anche ϵ .
E' obbligatorio che contenga un simbolo $\in V \setminus T$.

Due convenzioni:

1. I simboli non-terminali si indicano con CAPITAL letters, i simboli terminali si indicano con small letters.
2. La prima produzione ha come driver lo start symbol della grammatica.

Formalismo che indica come si deriva una qualche *parola* dalla grammatica:

data $\gamma \in V^*$, γ deriva direttamente da δ in $G \Leftrightarrow \gamma = \sigma \delta \tau$, $\delta \rightarrow \beta \in P$, $\gamma \in \sigma \beta \tau$ con $\sigma, \tau \in V^*$.

Esempio, linguaggio con 2 produzioni

$$\left\{ \begin{array}{l} S \rightarrow aSb \\ S \rightarrow ab \\ \hline a & b \end{array} \right. \quad \gamma = \underbrace{aaa}_{\sigma} \underbrace{S}_{\alpha} \underbrace{bbb}_{\tau} \quad \dots \text{cosa posso derivare?} \quad \text{Derivazione diretta: } \gamma = \underbrace{aaa}_{\sigma} \underbrace{ab}_{\beta} \underbrace{bbb}_{\tau}$$

vuota

Altra esempio: $\gamma = \underbrace{\omega}_{\sigma} \underbrace{S}_{\alpha} \underbrace{bbb}_{\tau}$, $\gamma = \underbrace{\omega}_{\sigma} \underbrace{ab}_{\beta} \underbrace{bbb}_{\tau}$

Il linguaggio di una grammatica è generato operando un passo di riscrittura dietro l'altro, fino a che nulla è più riscrivibile, ossia non esiste più alcuna stringa contenente simboli non-terminali che possa essere riscritta in una stringa di simboli terminali mediante una riduzione.

La stringa μ deriva da γ in G se esiste una sequenza di stringhe d_0, \dots, d_n tali che $\gamma = d_0, \mu = d_n, d_{i+1}$ deriva direttamente da d_i ($\forall i : 0 \leq i \leq n-1$).

$$\text{Ossia, } \begin{cases} S \rightarrow aSb \\ S \rightarrow ab \end{cases}$$

Supponiamo di partire con $\gamma = aaSbb$. Utilizzando la prima produzione, deriviamo $\mu = aaabbbb$ (μ deriva direttamente da γ).

- Utilizzando la seconda: $\mu' = aaaabbbb$ (derivazione da γ non diretta).

Il linguaggio generato dalla grammatica $G = (V, T, S, P)$ è definito come $L(G) = \{w \mid w \in T^* \text{ e } S \xrightarrow{*} w\}$

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

parole
Da S derivo in 1 o più passi

Se è lunga 0,
è la ϵ .
Stella: ammette
questo caso

Fosse stato $S \Rightarrow^* S$, ci sarebbe stata
derivazione $S \Rightarrow^* S$, che non serve a
derivare parole del linguaggio.

Esempi vari

$$G_1: S \rightarrow a \quad L(G_1) = \{a\}$$

$$G_2: S \rightarrow a \quad L(G_2) = \{a, b\}$$

$$S \rightarrow b$$

$$G_3: S \rightarrow B \quad L(G_3) = \{a\} \quad \text{La } B \text{ la posso riscrivere, ma non posso andare più avanti}$$

$$S \rightarrow a$$

$$G_4: S \rightarrow B \quad L(G_4) = \emptyset$$

$$G_5: S \rightarrow \epsilon \quad L(G_5) = \{\epsilon\} \quad \text{La parola vuota è una parola? Sì.}$$

Il linguaggio è un insieme che contiene la parola vuota, ma che non è vuoto.

$$G_6: S \rightarrow aSb \quad L(G_6) = \{a^n b^n \mid n \geq 0\}$$

$$S \rightarrow ab$$

$\xrightarrow{S \rightarrow ab}$
 $\xrightarrow{aSb \rightarrow a ab b}$
 $\xrightarrow{a ab b \rightarrow a aSbb}$
 $\xrightarrow{a aSbb \rightarrow a a ab bb}$
 \dots

$$G_7: S \rightarrow aSb \quad L(G_7) = \{a^n \cancel{b^n} \mid n \geq 0\}$$

$$S \rightarrow \epsilon$$

SUPERFLUO!

Ricordiamo che: 1) $a b = \epsilon a b, \epsilon a \epsilon b, \dots$ sono OMONIMI: "depuriamoli" da ϵ !

2) Elemento neutro moltiplicazione: \cdot

Elemento neutro concatenazione: ϵ

$$G_8: S \rightarrow aa b \quad S \xrightarrow{k} a^{2k} S b^k \quad k \geq 0$$

$$S \rightarrow A \quad S \xrightarrow{k} a^{2k} A b^k$$

$$A \rightarrow a A b b \quad A \xrightarrow{k,j} a^{2k} a^j A b^{2j} b^k \quad j \geq 0$$

$$A \rightarrow c \quad A \xrightarrow{k,j} a^{2k} a^j c b^{2j} b^k$$

$$L(G_8) = \{a^{2k+j} c b^{2j+k} \mid j, k \geq 0\}$$

Finora abbiamo visto grammatiche libere.

Eccone una non libera:

$$G_9: S \rightarrow aAb \quad L(G_9) = \{a^n b^n \mid n \geq 0\}$$

$$aA \rightarrow aaAb$$

$$A \rightarrow \epsilon$$

Nella precedente lezione abbiamo incontrato le grammatiche, dandone una definizione formale, e la nozione di derivazione, in uno e più passi.

Le grammatiche che abbiamo visto consistevano di più produzioni con lo stesso driver:

$$G: \begin{array}{l} S \rightarrow aSb \\ S \rightarrow ab \end{array} \quad \text{Possiamo condensare in: } S \rightarrow aSb \mid ab$$

La derivazione in più passi era di questo tipo:

$$\begin{array}{ll} G: & S \xrightarrow{\oplus} aAb \xrightarrow{\oplus} ab \\ & aA \rightarrow aaAb \\ & A \rightarrow \epsilon \end{array}$$

$$\begin{array}{c} a2A \xrightarrow{\oplus} a2bb \\ \Downarrow^{\oplus} \\ a2aAb \xrightarrow{\oplus} a2abb \\ \Downarrow^{\oplus} \\ a2a2Ab \xrightarrow{\oplus} a2abb \\ \Downarrow^{\oplus} \\ \dots \end{array}$$

$$L(G) = \{ a^n b^n \mid n \geq 0 \}$$

Oggi proviamo esercizi del tipo "dato il linguaggio, determina la grammatica".

$$L_1 = \{ a^n b^j \mid n \geq 0, j \geq 0 \}$$

Tentativo 1: errato

$$G_1: \begin{cases} S \rightarrow aSb \\ S \rightarrow b \end{cases} \quad \text{Abbiamo: } S \xrightarrow{\oplus} a^k S b^k, k \geq 0 \xrightarrow{\oplus} a^k b b^k, \text{ e cioè a } G_1 \text{ corrisponde } \{ a^n b^{n+1} \mid n \geq 0 \}$$

Tentativo 2: errato

$$G_2: \begin{cases} S \rightarrow aS \\ S \rightarrow b \end{cases} \quad \text{No, perché produce } L(G_2) = \{ a^n b \mid n \geq 0 \} \text{ e } a^2 b^2 \in L_1 \text{ e } a^2 b^2 \notin L(G_2)$$

Tentativo 3: errato

La mia proposta: CORRETTA!

$$G_3: \begin{cases} S \rightarrow ab \mid bB \\ A \rightarrow a \mid \epsilon \\ B \rightarrow b \mid \epsilon \end{cases} \quad \text{No, non c'è ricorsione sulla A}$$

$$S \rightarrow A \mid aS$$

$$A \rightarrow bA \mid b$$

Tentativo 4:

$$G_4: \begin{cases} S \rightarrow AB \\ A \rightarrow aA \mid a \mid \epsilon \\ B \rightarrow bB \mid b \end{cases}$$

superflua

E' corretto.

Abbiamo poi studiato i vari modi di ottenere 'a' da $A \rightarrow aA \mid a \mid \epsilon$

- 1 $A \Rightarrow aA \Rightarrow a$
- 2 $A \Rightarrow a$

È un esempio di grammatica ambigua: esiste almeno una parola appartenente al linguaggio per la quale si possono proporre due derivazioni "canoniche". Altamente negativo, perché non si analizzano (non si riesce).

L'analisi sintattica si pone come principale obiettivo di capire se il programma è derivabile dal linguaggio in cui lo ho scritto. Esistono varie tecniche per derivare le frasi, con l'albero, ad esempio; ma si vuole che questo albero per derivarle sia unico.

Facciamo un esempio: $1 + 2 \times 3 = 7 \dots$ ma perché non 9? Perché ci hanno insegnato le regole di precedenza. Proviamo a descrivere tale frase con la seguente grammatica:

$$S \rightarrow S + S \mid S \cdot S \mid n$$

$$S \stackrel{?}{\Rightarrow} * \text{num} + \text{num} \cdot \text{num} ?$$

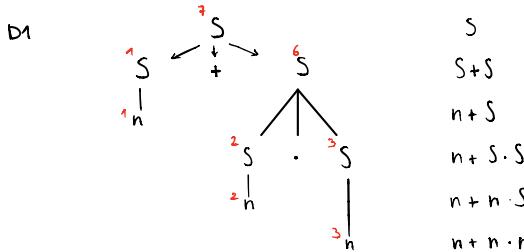
Scegliamo la convenzione rightmost

Individuiamo 2 derivazioni:

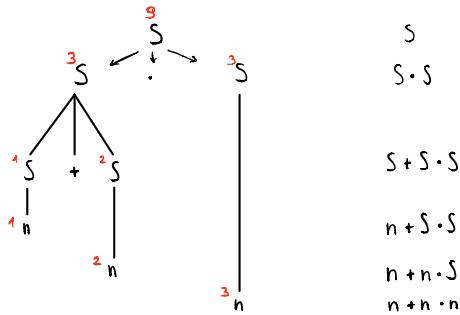
$$D1. \ S \xrightarrow{1} S+S \xrightarrow{2} \text{num}+S \xrightarrow{3} \text{num}+\text{S}\cdot\text{S} \xrightarrow{4} \text{num}+\text{num}\cdot\text{S} \xrightarrow{5} \text{num}+\text{num}\cdot\text{num}$$

$$D2. \ S \xrightarrow{1} S\cdot S \xrightarrow{2} S+S\cdot S \xrightarrow{3} \text{num}+S\cdot S \xrightarrow{4} \text{num}+\text{num}\cdot\text{S} \xrightarrow{5} \text{num}+\text{num}\cdot\text{num}$$

E saminiamone gli alberi di derivazione:



S
 $S+S$
 $n+S$
 $n+S\cdot S$
 $n+n\cdot S$
 $n+n\cdot n$



S
 $S\cdot S$
 $S+S\cdot S$
 $n+S\cdot S$
 $n+n\cdot S$
 $n+n\cdot n$

Eriamo partiti da $1 + 2 \times 3$. L'analisi lessicale aveva tradotto tale stringa in $\langle n, 1 \rangle + \langle n, 2 \rangle * \langle n, 3 \rangle$.

Abbiamo visto che è una grammatica ambigua; l'ambiguità non ci piace.

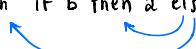
Definizioni:

- Una derivazione si dice **leftmost (rightmost)** se ad ogni passo viene sostituito il non-terminale più a sinistra (destra) della stringa corrente.
 - Una grammatica G è ambigua $\Leftrightarrow \exists w \in L(G)$ esistono 2 distinte derivazioni, entrambe leftmost oppure entrambe rightmost.
-

$$S \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } S \text{ else } S \mid a$$

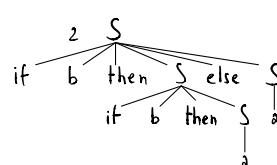
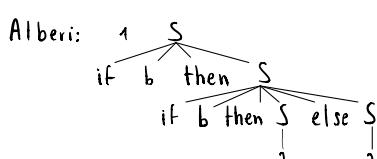
E' ambigua? Fornite un esempio.

FGAZ: if b then if b then a else a



$$1 - \text{if } b \text{ then } (\text{if } b \text{ then } a \text{ else } a)$$

$$2 - \text{if } b \text{ then } (\text{if } b \text{ then } a) \text{ else } a$$



Riflettiamo su $L: \{a^n b^n c^n \mid n > 0\}$

$$S \rightarrow a S B c \mid a b c$$

Proviamo: $S \xrightarrow{1} a S B c \xrightarrow{2} a a S B c B c \xrightarrow{3} a a S B B c c$

$$c B \rightarrow B c$$
$$b B \rightarrow b b$$

La scorsa lezione abbiamo visto che esistono più grammatiche che generano lo stesso linguaggio. Un esempio era la grammatica: $\{a^n b^n \mid n > 0\}$

Produzioni di una grammatica:

Contiene 1 non-terminale

$d \rightarrow P$

Può essere vuota: E

Def. Un linguaggio formale L è libero (da contesto) \Leftrightarrow se esiste una grammatica G libera (da contesto) tale che $L=L(G)$.

La grammatica $S \rightarrow aSB \mid abc$ produce il linguaggio $\{a^n b^n c^n \mid n > 0\}$. Non esiste una grammatica libera per questo linguaggio.

Come si fa a dimostrare che non c'è una certa grammatica libera che generi un linguaggio? Si dimostra con un lemma fondamentale.

Pumping lemma per linguaggi liberi

Sia L un linguaggio libero. Allora:

Indagare: $\exists p \in \mathbb{N}^+ \text{ t.c. } \forall z \in L, |z| > p, \exists u, v, w, x, y \text{ t.c. valgono:}$

$z = uvwxy$

$\begin{matrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{matrix}$

Anche nullo

Questo lemma serve dal punto di vista tecnico per organizzare una dimostrazione del fatto che un certo linguaggio *non* sia libero. Affinché un certo linguaggio non sia libero, si deve verificare la negazione della tesi:

$\# p \in \mathbb{N}^+, \quad \exists z \in L, |z| > p, \quad \# u, v, w, x, y \text{ t.c. valgono: } !(Q_1 \& Q_2 \& Q_3 \& Q_4)$

$$z = uvwxy \quad \& \quad |vwx| \leq p \quad \& \quad |vx| > 0 \Rightarrow \exists i \in \mathbb{N}, uv^i w^i y \notin L$$

De Maroan:

$$\begin{aligned} & ! (Q_1 \& Q_2 \& Q_3 \& Q_4) = \\ & ! Q_1 \text{ or } ! Q_2 \text{ or } ! Q_3 \text{ or } ! Q_4 = \\ & !(Q_1 \& Q_2 \& Q_3) \text{ or } ! Q_4 = \\ & Q_1 \& Q_2 \& Q_3 \Rightarrow Q_4 \end{aligned}$$

In altre parole:

Propongo una parola z , e dimostro che, anche se la scompongo in una qualsivoglia decomposizione di cinque parti che caratteristiche che abbiamo scritto, riesco a trovare un indice i che, applicato alla v e alla x , fa sì che la parola non appartenga ad L .

ESEMPIO: Proposizione: $\{a^n b^n c^n \mid n > 0\}$ non è un linguaggio libero.

DIMOSTRAZIONE:

Supponiamo che L sia un linguaggio libero. Sia p una costante arbitraria ($\#p \in \mathbb{N}^*$). Si consideri $z = a^p b^p c^p$ ($\exists z \in L, |z| > p$).

La parola che ho preso in considerazione è così fatta: a...a b...b c...c
P P P

Decomponendo la parola in $uvwxy$ con $|vwx| \leq p$ abbiamo le seguenti possibilità:

a a b b c c

Eiste questo i ? Certo che sì, ad esempio $i = 0$

$\Rightarrow \forall u, v, w, x \text{ se } z = uvwx \& |vwz| \leq p \& |vx| > 0, \text{ allora } vwx$ o non contiene occorrenze di 'c'
 $\Rightarrow uv^0wx^0y \notin L$, il che contraddice il pumping lemma. o non contiene occorrenze di 'a'

$$L_1 = \{ww \mid w \in \{a,b\}^*\}$$

$$\text{FGAZ: } z = a^p b^p a^p b^p$$

$$L_2 = \{w w^R \mid w \in \{a,b\}^*\}$$

Ex: abba

$$S \rightarrow aS_2 \mid bS_1 \mid \epsilon \checkmark$$

I linguaggi liberi sono chiusi per unione

L_1 ed L_2 linguaggi liberi $\Rightarrow L_1 \cup L_2$ è un linguaggio libero, ossia:

$\exists G_1, G_2$ grammatiche libere t.c. $L_1 = L(G_1)$ e $L_2 = L(G_2)$.

Esempio: $S \rightarrow aSb \mid ab \quad \{a^n b^n \mid n > 0\} > \{a^n b^n \mid n > 0\} \cup \{e^n d^n \mid n > 0\}$



TROVARE UNA GRAMMATICA

$$Z \rightarrow S \mid A \quad \xrightarrow{\text{ipotesi}} Z \notin V_1 \cup V_2$$

$$S \rightarrow aSb \mid ab$$

$$A \rightarrow cAd \mid cd$$

Sembra essere opportuna, ma c'è solo
un dettaglio di cui ci dobbiamo occupare

$$\begin{array}{ll} \{a^n b^n \mid \dots\} & S \rightarrow aSb \mid ab \\ \{c^n d^n \mid \dots\} & S \rightarrow cSd \mid cd \end{array} \quad Z \rightarrow S \mid S \quad \text{genererebbe un macello: necessità di fare d-equivalenza!}$$

(Riscrivere i non-terminali ononimi)

$$G = (\{Z\} \cup V_1 \cup V_2^{\downarrow}, T_1 \cup T_2, Z, \{Z \rightarrow S_1 | S_2^{\downarrow}\} \cup P_1 \cup P_2^{\downarrow})$$

Nuovo start symbol

- $V_2^{\downarrow}, S_2^{\downarrow}, P_2^{\downarrow}$ ottenuti da V_2, S_2 e P_2 per ridenominazione dei non-terminali ononimi a quelli di $V_1 \setminus T_1$
- $Z \notin V_1 \cup V_2^{\downarrow}$

G è tale che $L(G) = L(G_1) \cup L(G_2)$

I linguaggi liberi sono chiusi per concatenazione

L_1 ed L_2 concatenati fanno $\{w_1 w_2 \mid w_1 \in L_1 \& w_2 \in L_2\}$

DIMOSTRAZIONE L_1, L_2 linguaggi liberi $\Rightarrow \exists G_1 = (V_1, T_1, S_1, P_1)$ & $G_2 = (V_2, T_2, S_2, P_2)$ libere e tali che $L_1 = L(G_1)$ & $L_2 = L(G_2)$

Sia $G_2' = (V_2', T_2', S_2', P_2')$ una d-ridenominazione di G_2 tale che evita possibili clash con i non-terminali di G_2 .

Allora $G = (\{Z\} \cup V_1 \cup V_2', T_1 \cup T_2, Z, P_1 \cup P_2' \cup \{Z \rightarrow S_1 | S_2'\})$ è tale che $Z \notin V_1 \cup V_2'$

$L(G)$ è la concatenazione di L_1 e L_2 .

I linguaggi liberi non sono chiusi per intersezione

L_1, L_2 liberi ... $L_1 \cap L_2$ libero?

Esempio: $L_1 = \{a^n b^n c^j \mid n, j > 0\} \rightarrow$ Libero! Nota che c'è la concatenazione di $\{a^n b^n \mid n > 0\}$ e $\{c^j \mid j > 0\}$.

$$L_2 = \{a^j b^n c^n \mid n, j > 0\} \rightarrow$$
 Libero.



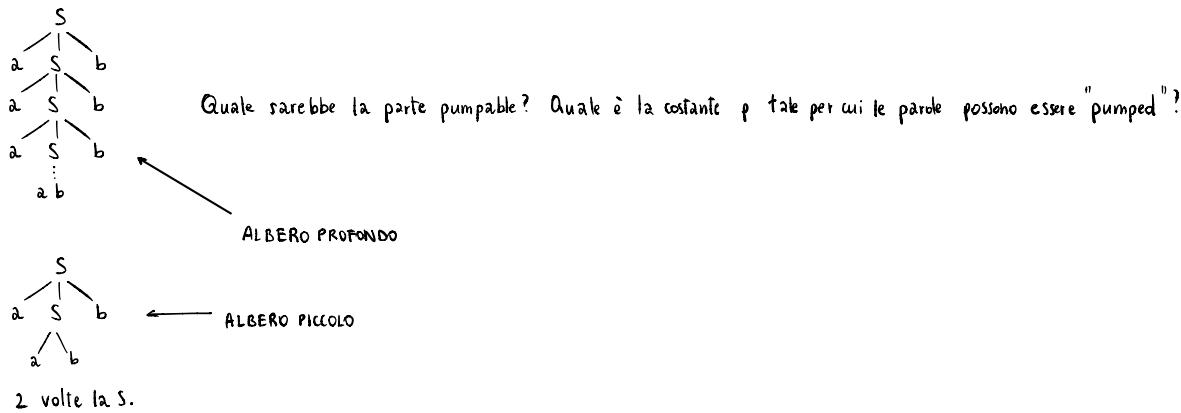
$$L_1 \cap L_2 = \{a^n b^n c^n \mid n > 0\} \rightarrow \text{NON E' LIBERO!}$$

Considera la seguente grammatica: $G: S \rightarrow a$
 $L(G) = \{a\}$ c'è qualcosa di pumpable? No, cioè $p=1$

Role

$S \sim a S b \mid a b \quad L(G) = \{a^n b^n \mid n > 0\}$ Qui sì, c'è qualcosa di pumpable.

Mostriamo l'albero di derivazione associato ad una certa parola:

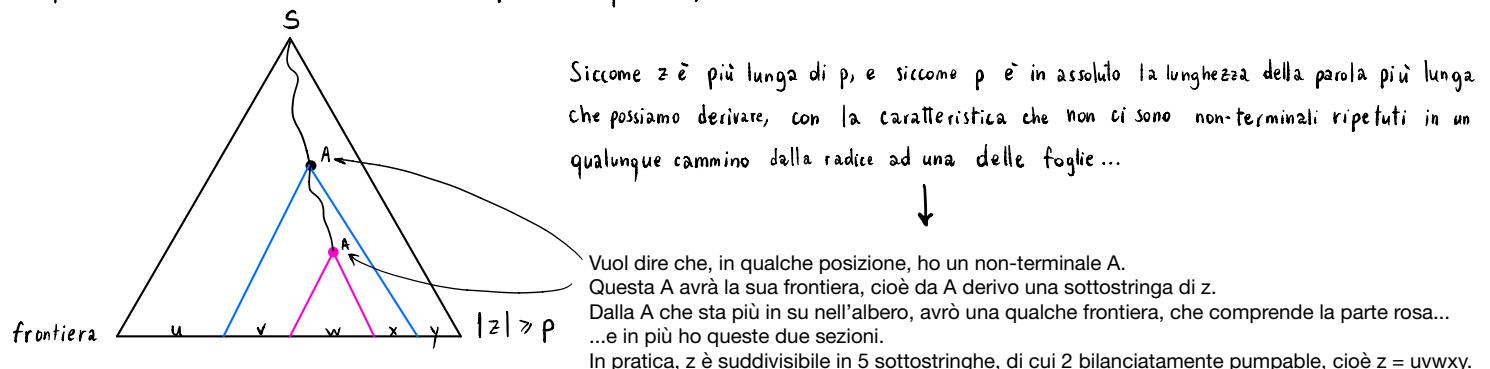


La costante p cui si fa riferimento nel pumping lemma è la lunghezza della parola in assoluto più lunga che uno può derivare mediante un albero di derivazione in cui, lungo ogni possibile cammino dell'albero, non c'è mai lo stesso non-terminale ripetuto.

Se ad un linguaggio appartiene una parola la cui lunghezza è maggiore di p , significa che quella parola si deriva mediante un albero in cui c'è almeno un cammino, dalla radice alla foglia, in cui un non-terminale è ripetuto due volte.

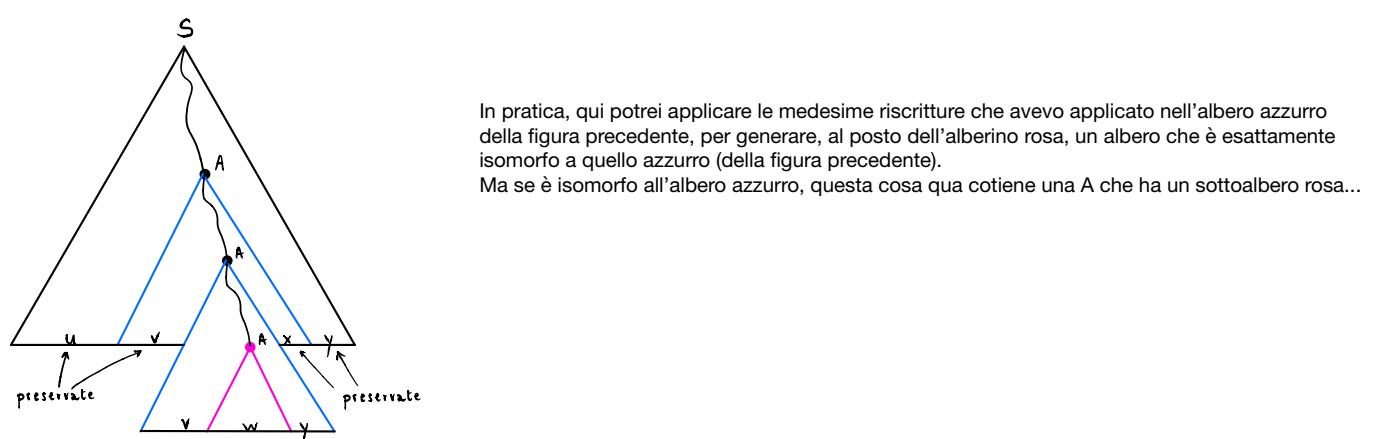
Che significa questo?

Sia questo l'enorme albero con cui deriviamo la parola in questione, z :



...E perché è pumpable?

Se io ho due alberi, l'albero azzurro e l'albero rosa, che comunque entrambi hanno una radice A , non posso escludere una derivazione fatta così:



Questo "scherzettino" qui lo posso ripetere tutte le volte che voglio; naturalmente anche 0 volte (che corrisponde a dire che questa A qua si espande secondo le derivazioni che rappresentano il sottoalbero rosa).

Se rappresento un albero rosa, vuol dire che alla mia frontiera mancherà la parte v e la parte x (pumping per $i=0$).

Lo scherzettino lo posso ripetere, e in generale ho:

Se la parola è $>$ di p , e p è questa costante che dice quanto è lunga la parola in assoluto più lunga che possiamo derivare, con la caratteristica che non ci sono non-terminali ripetuti in un qualunque cammino dalla radice ad una delle foglie, allora sicuramente c'è un cammino in cui è ripetuto un non-terminale, e da questo cammino nasce la porzione $z = u [vwx] y$, mentre la w nasce esattamente dal sottoalbero che è radicato nel punto in cui ho ripetuto il non-terminale. E allora, queste 2 parti le posso ripetere tante volte quante voglio.

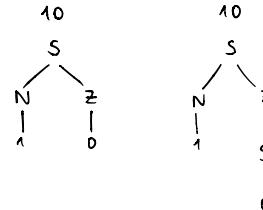
Senza troppi dettagli, del tipo "perché v e x non sono entrambe nulle?"...

La grammatica che ho proposto: Non va bene: è fortemente ambigua. Esempi:

$$S \rightarrow Nz | \varepsilon | 0$$

$$N \rightarrow 1 | \dots | 9 | NN$$

$$Z \rightarrow 0 | ZZ | S$$



Ambigua → indecidibile

Una grammatica che ha lo stesso problema è:

$$S \rightarrow N$$

$$N \rightarrow 1 | \dots | 9 | Z | \varepsilon$$

$$Z \rightarrow 0 | N$$

Vediamo due grammatiche corrette:

$$\text{FGAZ} \quad S_1 \rightarrow 0 | 1N | \dots | 9N$$

$$N \rightarrow \varepsilon | 0N | \dots | 9N$$

$$S_2 \rightarrow N | NA | 0$$

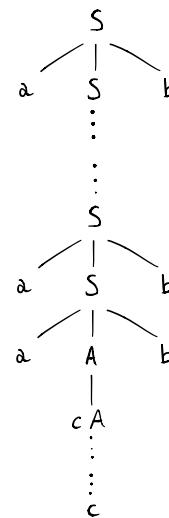
$$N \rightarrow 1 | \dots | 9$$

$$A \rightarrow 0A | \dots | 9A | N | 0$$

PROFESSORESSA: $\{a^n c^j b^n \mid n, j > 0\}$ è libero? Se sì, datemene una grammatica.

$$S \rightarrow aSb | aAb$$

$$A \rightarrow c | cA$$



Oggi introdurremo l'analisi lessicale.

Vediamo innanzitutto le **espressioni regolari**. Le seguenti regole definiscono le *espressioni regolari di un alfabeto* Σ .

Sia Σ un alfabeto, cioè un insieme di simboli. Da Σ si definiscono in maniera induttiva le espressioni regolari. Si dice cioè quali sono le minime espressioni regolari, gli operatori che possiamo usare fra tali espressioni; utilizzando espressioni regolari e operatori possiamo comporre espressioni più grandi.

La specifica di un'espressione regolare è un'esempio di definizione ricorsiva, in quanto fa uso di base e passo induttivi:

Base:

- 1 Σ è un'espressione regolare, che denota il linguaggio $\{\Sigma\}$.
 2 Se a è un simbolo di Σ , allora a è un'espressione regolare che denota il linguaggio $\{a\}$.

Passo:

Siano r , s due espressioni regolari, che rispettivamente denotano i linguaggi $L(r)$ e $L(s)$.

Allora valgono

1. $r \mid s$ è un'espressione regolare che denota $L(r) \cup L(s)$.
 2. $r \cdot s$ (meno comunemente, $r \cdot s$) è un'espressione regolare che denota $L(r)L(s) := \{w = w_1w_2 \text{ & } w_1 \in L(r) \text{ & } w_2 \in L(s)\}$
 3. r^* è un'espressione regolare che denota $L(r^*) = \{\varepsilon\} \cup \{w_1 \dots w_k \mid \#_i = 1, \dots, k, w_i \in L(r)\}$.
 4. (r) è un'espressione regolare che denota $L(r)$.

Esempio: $L(a) = \{a\}$, $L((a)) = \{a\}$.

Le parentesi vengono utilizzate per meglio evidenziare l'ordine di precedenza e associatività.

Convenzioni su precedenza e associatività degli operatori

- * Ha precedenza più alta
 - Ha precedenza inferiore a *
 - | Ha precedenza inferiore a .

} associativi a sinistra

Esempio: $a \mid b^*c$ 1. $a \mid (b^*)c \rightarrow$ 2. $a \mid ((b^*)c)$

$$\begin{aligned}
 \text{Sia } r = a \mid b^* c. \quad L(r) &= L(a) \cup L(b^* c) \quad \cdot L(b^* c) = \{w \mid w = w_1 w_2 \text{ & } w_1 \in L(b^*) \text{ & } w_2 \in L(c)\} \\
 &\quad \cdot L(c) = \{c\} \Rightarrow w_2 = c \\
 &= \{wc \mid w \in L(b^*)\} \\
 &= \{a\} \cup \{b^n c \mid n \geq 0\}
 \end{aligned}$$

Facciamo alcuni esempi

r	$L(r)$
$a \mid b$	$\{a, b\}$
$ab \mid b$	$\{ab, b\}$
$a(blc)$	$\{ab, ac\}$
a^*b^*	$\{a^n b^j \mid n, j \geq 0\} \rightarrow$ si può fare di meglio: $\{a^n \mid n \geq 0\}$
$a \mid a^*b$	$\{a\} \cup \{a^n b \mid n \geq 0\}$

Ora proviamo il contrario.

L r
Identifieri, cioè stringhe, che cominciano $(a|...|z)(a|...|z|0|...|g)^*$
con una lettera dell'alfabeto, cui seguono
lettere o numeri.

Numeri binari multipli di 2

✓ $(0|1)^*$ 0 E' possibile che cominci con 0, ma per la prof vale "full mark"

✓ $1(0|1)^* 0$ Va bene... dipende se vogliamo considerare zero un multiplo di 2 o no

Vorremmo aggiungere la possibilità "0": $(1(01)*)$

Stringhe di 'a' e di 'b' con 'a' consecutive

$\vee (a|b)^* \wedge (a|b)^*$

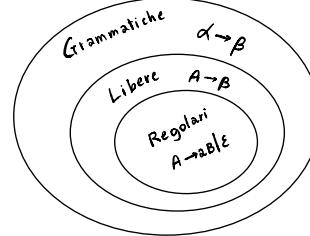
77 Senza 78

senza

MANC

~~(ab*|b)*~~ NO, "AA" \notin L(r)

L'argomento di oggi è ricavare un NFA a partire da un'espressione regolare.
Vedremo che il linguaggio denotato dall'espressione regolare è lo stesso di quello riconosciuto dall'NFA.



Dato un minDFA, possiamo ricavare l'espressione regolare, ma non lo vedremo perché è difficile.

NFA: Non-deterministic Finite-state Automata: si definiscono in maniera formale con una tupla:

$$(S, A, \text{move}_n, s_0, F)$$

S : insieme di stati

A : alfabeto: insieme di simboli. $\epsilon \notin A$

s_0 : uno stato $\in A$ iniziale

F : insieme di stati finali $\subseteq S$

move_n : funzione di transizione: $S \times (A \cup \{\epsilon\}) \rightarrow P(S)$ Dato uno stato e un elemento dell'alfabeto o ϵ , restituisce un insieme di stati. → Da un certo stato s ad una certa etichetta ($A \cup \{\epsilon\}$)

Insieme delle parti = insieme dei sottoinsiemi

Convenzioni per la rappresentazione grafica delle strutture di un NFA

NFA: rappresentato da un grafo in cui:

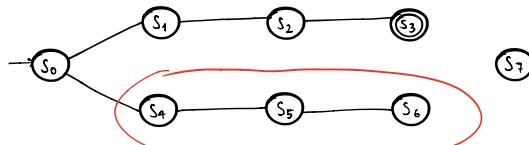
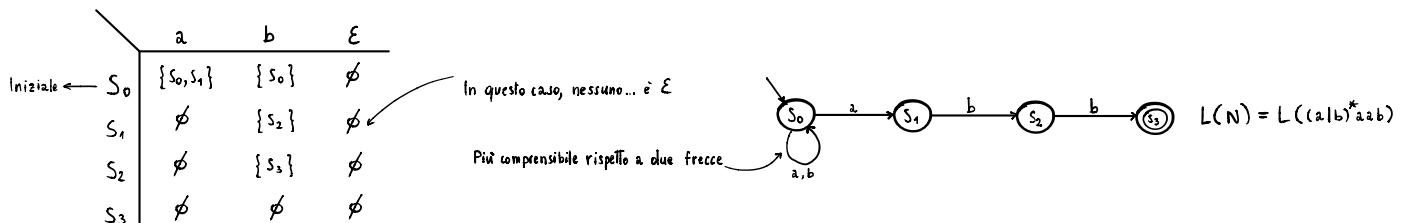
NODI = STATI

ARCHI = move_n = funzione di transizione; sono etichettati da elementi dell'insieme $A \cup \{\epsilon\}$. Cioè, ad ogni arco, è associata un'etichetta, che è una parola dell'alfabeto o ϵ .

Il nodo che rappresenta s_0 si evidenzia con una piccola freccia entrante.

I nodi che rappresentano stati finali in F sono disegnati con un doppio cerchio.

ESEMPIO



Attenzione!

Non c'è scritto che tutti i cammini portano allo stato finale, né che tutti gli stati siano raggiungibili.

Non contribuisce, in quanto non raggiunge uno stato finale

L'analizzatore lessicale costruisce un'espressione regolare che denota l'insieme di tutti e quanti gli identificatori, e da questo, mediante un algoritmo (che esiste in più varianti), costruisce l'NFA che accetta il linguaggio denotato da quell'espressione regolare.

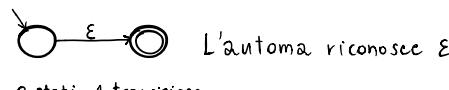
Costruzione di Thompson

Data un'espressione regolare r , costruisce un NFA N tale che $L(r) = L(N)$.

La dimostrazione è induttiva. La costruzione è interessante, in quanto per come è organizzata, ad ogni passo si introducono al massimo due nuovi stati.

Ogni NFA intermedio ottenuto durante la costruzione ha esattamente uno stato finale, non ha archi entranti nello stato iniziale, non ha archi uscenti dallo stato finale.

Base • $r = \epsilon, L(r) = \{\epsilon\}$



L'automa riconosce ϵ .

2 stati, 1 transizione

• $r = a \in A, L(r) = \{a\}$



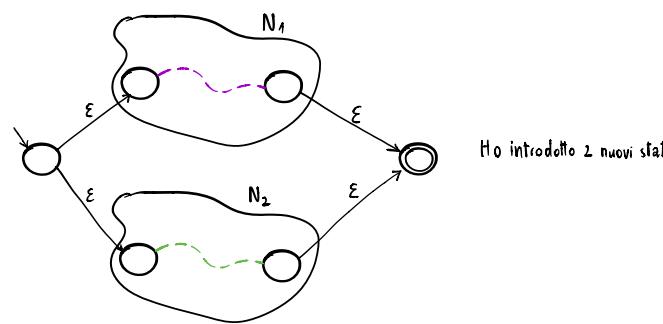
E NON:

che sarebbe a^*

Passo Immagino di avere 2 automi, N_1 t.c. $L(N_1) = L(r_1)$ e N_2 t.c. $L(N_2) = L(r_2)$, dati per ipotesi induttiva

Ora vogliamo dire come costruiamo i seguenti automi:

$r_1 | r_2$



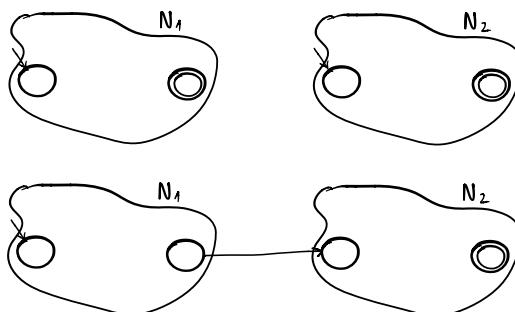
Se $w_i \in L(r_i)$,

...Analogamente

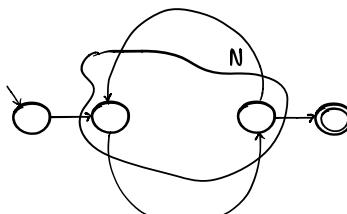
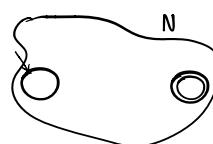
- ⇒ ∃ un cammino etichettato da s_{0i} a s_{fi} ,
- ⇒ $\epsilon w_i \epsilon$ è lo spelling di un cammino da s_0 a s_f
- ⇒ $w_i \in L(N)$

$(r_1), L((r_1))$ lo ho già per definizione

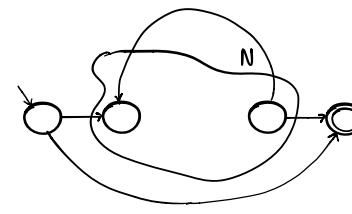
$(r_1 \cdot r_2), L(r_1 \cdot r_2)$



r_1^*



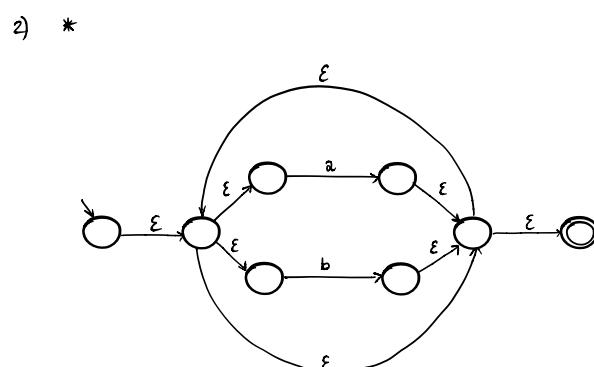
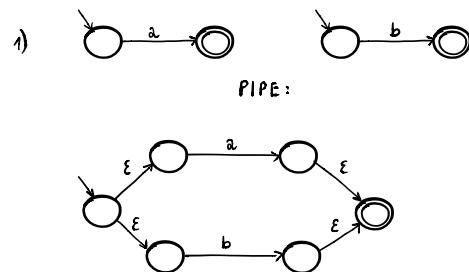
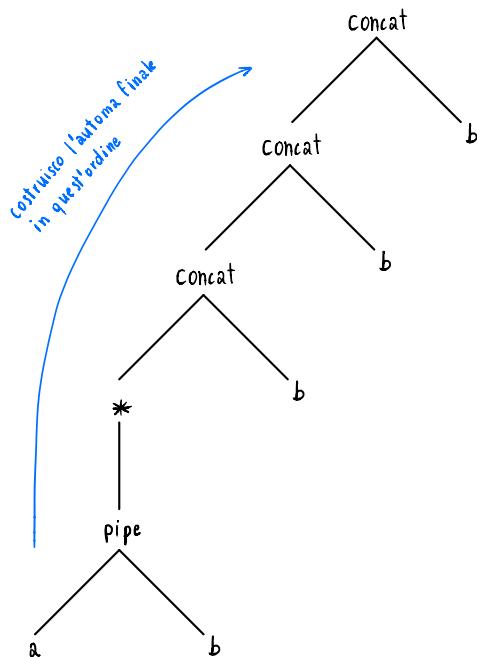
Versione di Giulia



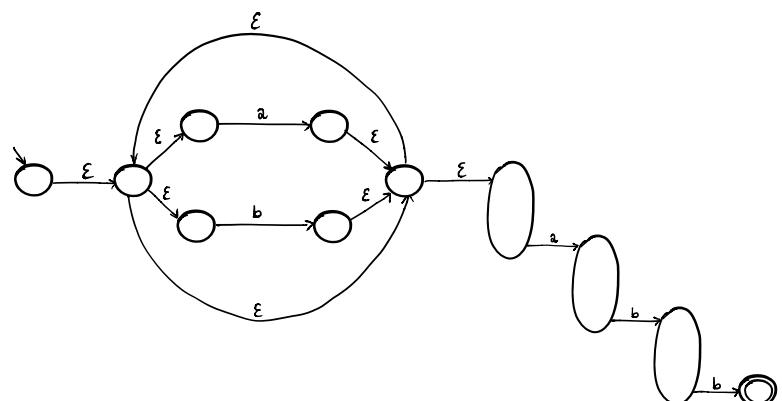
Libro

ESEMPIO

$(a \mid b)^* abb$



3) Varie concatenazioni: con a, b, b



Domande che ho fatto alla professoressa alla fine della lezione:

- Per un certo linguaggio, dobbiamo identificare l'NFA più compatto?

No, finora non ci siamo occupati di questo. Infatti, per un certo linguaggio possiamo realizzare più di un NFA.

- Ma, non dovremmo garantire che ci sia solo un modo per generare una stringa?

No, gli NFA di cui abbiamo finora parlato, ammettono potenzialmente più cammini. L'unico vincolo è che almeno uno esista.

Abbiamo visto, in ordine, le espressioni regolari; poi, come con la costruzione di Thompson, si possa ottenere un NFA che riconosca esattamente il linguaggio denotato da quell'espressione regolare.

Oggi vedremo la simulazione di un NFA, cioè useremo l'NFA per verificare se accetta una certa parola che gli diamo in input.

Un NFA N **accetta** la parola w , se e solo se esiste un cammino $x_1 \dots x_k = w$ che parte dallo stato iniziale e termina in uno stato finale. La definizione mette in luce diversi gradi di non-determinismo: per una stessa parola, vi possono potenzialmente essere più di un cammino etichettato. Se eventualmente vogliamo identificare tutti i cammini, possiamo adottare due strategie. Una consiste nell'utilizzare una tecnica di backtracking. Esiste tuttavia una strategia più efficiente, che ha a che fare con la ϵ -closure.

Supponiamo di avere l'NFA $N = (S, A, move_n, s_0, F)$ con $t \in S$

La ϵ -closure($\{t\}$) è l'insieme degli stati in S che possono essere raggiunti da t tramite 0 o più ϵ -transizioni (transizione etichettata da ϵ).



Fondamentale: la ϵ -closure di $\{t\}$ può essere vuota? NO, $t \in \epsilon$ -closure($\{t\}$)

$T \subseteq S$ ϵ -closure(T) = $\bigcup_{t \in T} \epsilon$ -closure($\{t\}$) ossia:

- tutti gli stati in T

- e tutti gli stati che possono essere raggiunti da essi con ϵ -transizioni.

COMPUTAZIONE DI ϵ -closure($\{t\}$)

$N = (S, A, move_n, s_0, F)$

$t \in S$

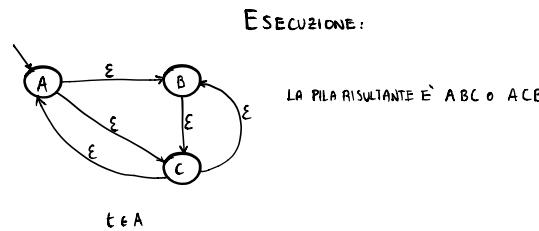
STRUTTURE DATI DI CUI CI AVVALIAMO:

- Pila: per le computazioni parziali
- Array booleano "alreadyOn" per sapere in tempo costante se gli stati sono nella pila
- Array bidimensionale per memorizzare la funzione $move_n$, di dimensione $|S| \times |A| + 1$ in cui ogni entry (t, x) è una lista linkata che rappresenta $move_n(t, x)$.

colonna di ϵ

ALGORITMO

```
alreadyOn = {FALSE ... FALSE};    %. INIZIALIZZAZIONE
closure (t, stack) {
    stack.push(t);
    alreadyOn[t] = TRUE;
    for (stato u ∈ move_n(t, ε)) {
        if (!alreadyOn[u])
            closure (u, stack);
    }
}
```



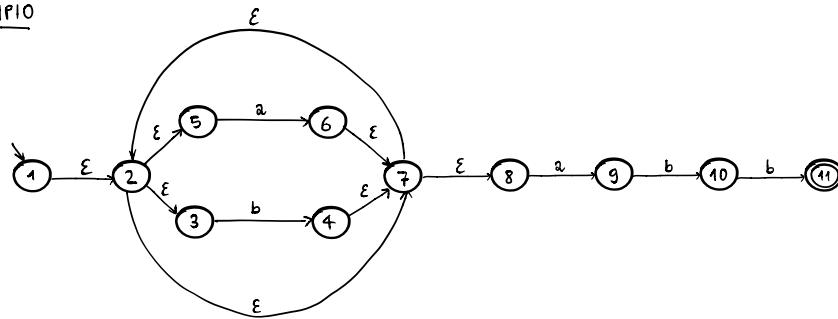
SIMULAZIONE DI NFA

INPUT: $w \in \$$ con $\$ \notin A$, NFA $N = (S, A, move_n, s_0, F)$

OUTPUT: YES/NO se $w \in L(N)$ o $w \notin L(N)$

1. $states = \epsilon$ -closure($\{s_0\}$)
2. $symbol = nextchar()$ %. input buffer
3. while ($symbol \neq \$$) {
4. $states = \epsilon$ -closure($\bigcup_{t \in states} move_n(t, symbol)$) %. Per ogni stato $t \in states$, guardo dove mi muovo col symbol che sto leggendo
5. $symbol = nextchar()$
6. }
7. if ($states \cap F \neq \emptyset$) return YES; else return NO

ESEMPIO



$w = abaabb \in \text{linguaggio?}$

INPUT: $w\$ = abaabb\$$

ESECUZIONE:

```

1. states = ε-closure ({s0})
2. symbol = nextchar()
3. while (symbol ≠ \$) {
4.   states = ε-closure (Utεstates moven(t, symbol))
5.   symbol = nextchar()
6.   while (symbol ≠ \$) {
7.     states = ε-closure (Utεstates moven(t, symbol))
    :
7.   if (states ∩ F ≠ ∅) return YES; else return NO      YES
  
```

$\text{states} = \text{ε-closure}(\{1\}) = \{1, 2, 3, 5, 7, 8\}$

$\text{symbol} = a$

TRUE

$\text{states} = \text{ε-closure}(\{6, 9\}) = \{6, 9, 7, 8, 2, 5, 3\}$

$\text{symbol} = b$

TRUE

$\text{states} = \text{ε-closure}(\{10, 4\}) = \{10, 4, 7, 8, 2, 5, 3\}$

Sicuramente c'è perché lo sto chiudendo

Alla fine, otteniamo YES. YES indica che negli states ce ne è almeno uno finale, e quindi che c'è un cammino che porta allo stato finale.

Vi sono due approcci per l'utilizzo degli NFA:

- Utilizzarlo direttamente come riconoscitore di un linguaggio (pesante, perché deve calcolare ogni volta le chiusure);
- Trasformarlo in un DFA equivalente (che riconosca il medesimo linguaggio), che consente un'analisi più veloce (lineare). Spesso non vale la pena di fare questa trasformazione. È il caso di grep, in cui non è prevista una forma di riciclaggio della chiusura che si costruisce.

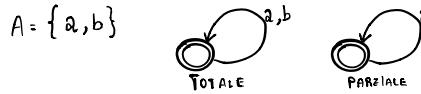
Nella scorsa lezione abbiamo visto l'NFA. L'NFA, data una certa parola, supporta un algoritmo la cui esecuzione specifica se una certa parola è riconosciuta da quell'NFA, e quindi se appartiene al linguaggio denotato dalla regular expression.

Una struttura alternativa all'NFA, ma equivalente, è il DFA, che sta per Deterministic Final-state Automaton. In che senso i DFA sono un sottoinsieme degli NFA?

- Innanzitutto, non ci sono ϵ -transizioni.
- La funzione di transizione può essere totale o parziale (questa differenza non è sempre messa in evidenza dai libri, come il "dragon book").

Funzione di transizione totale: dato un qualunque stato dell'automa che stiamo considerando, e dato uno qualunque dei simboli dell'alfabeto, la funzione di transizione è sempre definita; ossia c'è sempre una transizione uscente etichettata da quel simbolo.

Parziale: la funzione di transizione non è necessariamente definita per tutti i simboli.



Nuovamente:

Se il DFA ha funzione di transizione totale, allora, per ogni stato s dell'automa, e per ogni simbolo b dell'alfabeto, c'è esattamente una transizione uscente da s ed etichettata b .

Se ha funzione di transizione parziale, allora, per ogni stato s dell'automa, e per ogni simbolo b dell'alfabeto, c'è al più una transizione uscente da s ed etichettata b .

Definizione formale di DFA: tupla $(S, A, \text{move}_d, s_0, F)$

S : insieme degli stati

A : alfabeto: insieme dei simboli per cui avremo transizioni

move_d : funzione di transizione, cioè che, dato un elemento dell'alfabeto restituisce uno stato. $S \times A \rightarrow S$

s_0 : stato finale, $\in S$.

F : insieme degli stati finali, $\subseteq S$.

funzione "parziale"
notazione per noi superflua
↓

Le ϵ transizioni sono in realtà ammesse per il seguente corner-case:

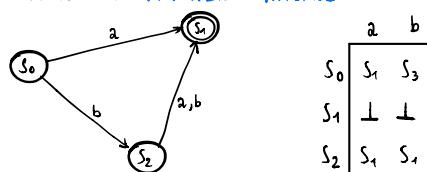
- $\epsilon \in L(D)$ se $s_0 \in F$ (se s_0 è finale, allora $\epsilon \in$ linguaggio).

Oltre tutto abbiamo:

- $w \neq \epsilon \in L(D)$ se $\exists!$ un cammino $x_1 \dots x_k = w$ da s_0 a uno stato $s \in F$
se esiste, è unico (struttura totalmente deterministica \rightarrow non più come l'NFA)

Convenzione: se move_d non è definita sul punto (s, b) , ossia ho uno stato s da cui non c'è una transizione uscente etichettata b , scriviamo $\text{move}_d(s, b) = \perp$ ("bottom")

SIMULAZIONE DFA CON FUNZIONE DI TRANSIZIONE PARZIALE



	a	b
s_0	s_1 s_3	
s_1	\perp \perp	
s_2	s_1 s_1	

ALGORITMO:

INPUT: $w \$$, DFA $D = (S, A, \text{move}_d, s_0, F)$

OUTPUT: sì/no alla domanda " $w \in L(D)$ "

1. $\text{state} = s_0$
 2. $\text{symbol} = \text{nextchar}()$
 3. $\text{while } (\text{symbol} \neq \$) \& (\text{state} \neq \perp) \{$
 4. $\text{state} = \text{move}_d(\text{state}, \text{symbol})$
 5. $\text{symbol} = \text{nextchar}()$
 6. }
 7. $\text{if } (\text{state} \in F) \text{ return YES; else return NO}$
- % cominciamo posizionandoci sullo stato iniziale dell'automa
% punta al primo elemento di $w \$$
% se la funzione di transizione è totale, omettiamo questo pezzo.
Totale = definita in tutti i punti, il bottom è completamente inutile.
% siamo arrivati allo stato finale ?

Costi:

- Simulazione NFA: $O(|w| \cdot (n+m))$
- Simulazione DFA: $O(w)$

Ma quanto costa passare dall'NFA al DFA equivalente (che riconosce lo stesso linguaggio)? Idea: utilizzare la ϵ -chiusura introdotta nella simulazione di NFA per far corrispondere sottoinsiemi di stati di NFA ad un unico stato del DFA.

ALGORITMO DI SUBSET CONSTRUCTION (C'è sempre nel compito!)

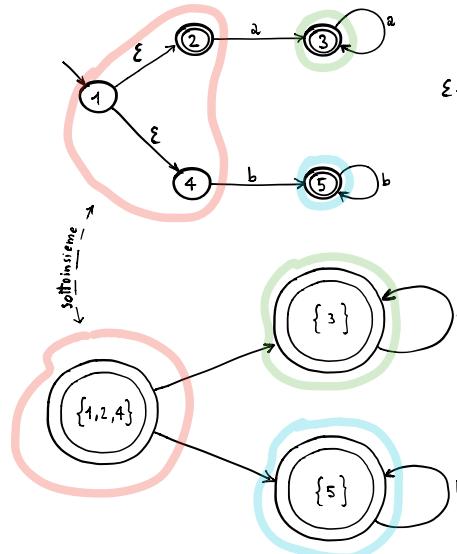
INPUT: NFA = $(S^n, A, move_n, s_0^n, F^n)$ $n :=$ "non-deterministico"
 INPUT: DFA = $(S^d, A, move_d, s_0^d, F^d)$ $d :=$ "deterministico"

```

1    $s_0^d = \epsilon\text{-closure}(\{s_0^n\})$ 
2    $S^d = \{s_0^d\}$ 
3   flag  $s_0^d$  come "non marcato"
4   while ( $\exists t \in S^d$  t.c.  $t$  non è marcato) {
5       flag  $t$  come "marcato"
6       foreach ( $b \in A$ ) {
7            $t' = \epsilon\text{-closure}(\bigcup_{t_i \in t} move_n(t_i, b))$ 
8           if ( $t' \neq \emptyset$ ) {
9                $move_d(t, b) = t'$ 
10              if ( $t' \notin S^d$ ) {
11                  aggiungi  $t'$  a  $S^d$ 
12                  flag  $t'$  come "non marcato"
13              }
14          }
15      }
16  }
17  foreach ( $t \in S^d$ ) {
18      if ( $t \cap F^n \neq \emptyset$  then  $t \in F^d$ 
19  }

```

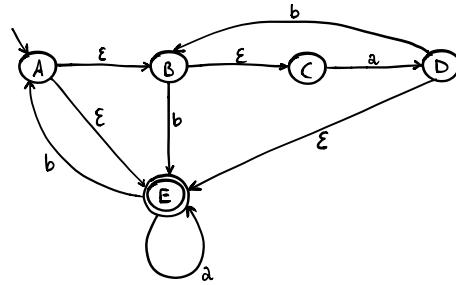
ESEMPIO (RAPIDO)



Scrivo direttamente la tabella di transizione:

ϵ -closure di $\{1\}$	ELEMENTI DI A	
	a	b
iniziale $\{1,2,4\}$	$\{3\} = t'$	$\{5\}$
finale $\{3\}$	$\{3\}$	\emptyset
finale $\{5\}$	\emptyset	$\{5\}$

ESEMPIO (ESTESO)



E' la ϵ -chiusura dello stato iniziale non-deterministico che mi dà lo stato iniziale dell'automa deterministico

La tabella di transizione sarà così fatta:

ELEMENTI DI A	STATI	
	a	b
iniziale	{A, B, E, C}	

↑
Non è marcato

↓ ↓

① a-transizioni:

La move_d sarà definita da uno stato che contiene la ϵ -chiusura di **B** e di **D**. Quale sarà questo stato? Con la ϵ posso andare comunque solo in **E** e in **D**.

ELEMENTI DI A	STATI	
	a	b
{A, B, E, C}	{E, D}	

Non vado da nessuna parte

② Il while è finito; torno a vedere se c'è uno stato che non è marcato. Sì, è {E, D}

Per lui devo vedere quali sono i possibili move_d rispetto alla a-transizione.

ELEMENTI DI A	STATI	
	a	b
{A, B, E, C}	{E, D}	{E, A, B, C}
{E, D}	{E}	
{E}		

ϵ -chiusura di {E}, che non aggiunge nulla.

L'insieme che contiene solo la E non lo ho ancora elencato:

lo aggiungo alla collezione.

⑤

ELEMENTI DI A	STATI	
	a	b
{A, B, E, C}	{E, D}	{E, A, B, C}
{E, D}	{E}	{A, B, C, E}
{E}	{E}	

a
E

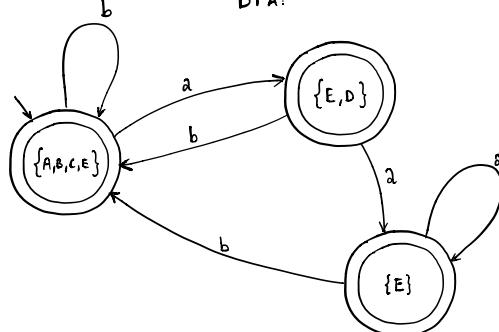
⑥

ELEMENTI DI A	STATI	
	a	b
{A, B, E, C}	{E, D}	{E, A, B, C}
{E, D}	{E}	{A, B, C, E}
{E}	{E}	{A, B, C, E}

b
EA

Mi resta da determinare quali saranno stati finali nel DFA. Nell'NFA l'unico stato finale è E. Nel DFA, tutti i sottoinsiemi elencati che contengono la E sono finali.

DFA:



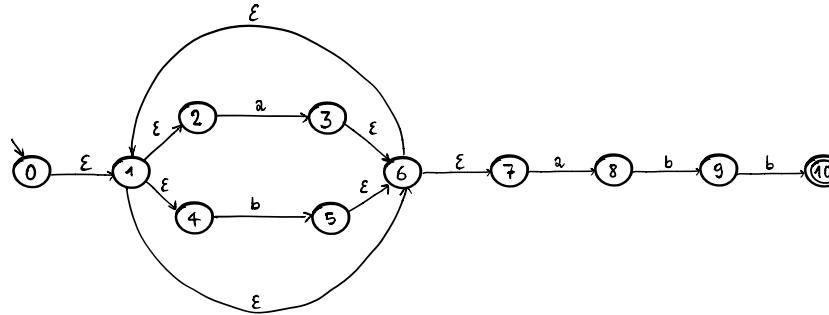
Abbiamo visto, studiando l'analisi lessicale:

- Definizione induttiva delle espressioni regolari. $1 \rightarrow 3$: costruzioni di Thompson
- Definizione di NFA come riconoscitore di un linguaggio denotato da un'espressione regolare.
- Definizione di DFA, e trasformazione di NFA in DFA mediante subset construction.

Oggi: un po' di pratica sulle costruzioni.

Espressione regolare $r = (a|b)^* a bb$

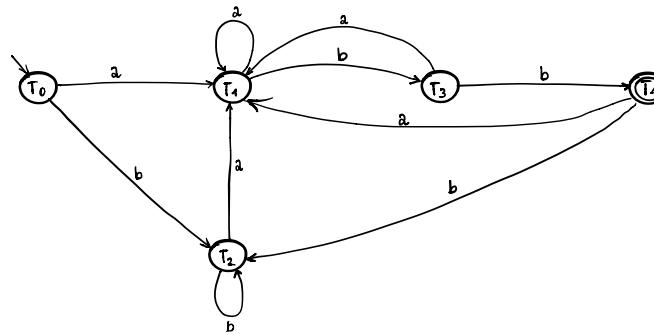
Il suo NFA:



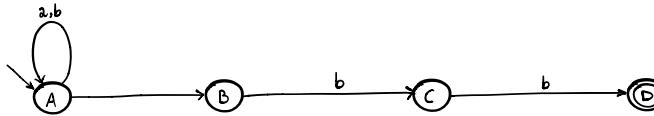
Proviamo a fare la subset construction dell'NFA per ottenere il corrispondente DFA.

a	b
$T_0 = \{0, 1, 2, 4, 6, 7\}$	T_1
$T_1 = \{3, 8, 6, 1, 2, 4, 7\}$	T_2
$T_2 = \{5, 6, 1, 2, 4, 7\}$	T_3
$T_3 = \{5, 9, 6, 1, 2, 4, 7\}$	T_4
$T_4 = \{5, 10, 6, 1, 2, 4, 7\}$	T_5

Disegniamo il DFA:



Vediamo un NFA alternativo per lo stesso linguaggio:



Domanda: "È normale che l'NFA sia più piccolo in memoria rispetto al DFA?". Memoria = quantità di nodi e di archi.

Proprietà del DFA minimo

Sia D un DFA con funzione di transizione totale (un DFA che ha funzione di transizione totale è, ad esempio, quello dell'esercizio prima proposto).

Abbiamo $\text{min}(D) = \text{subset}(\text{reverse}(\text{subset}(\text{reverse}(\text{reverse}(D)))))$.

Per ottenere il reverse DFA è sufficiente, detto grezzamente, invertire ogni arco; inoltre, lo stato iniziale diventa quello finale, e quello finale diventa quello iniziale.

Un linguaggio L si dice *regolare* se: (4 condizioni **equivalenti**).

esiste un'espressione regolare r t.c. $L = L(r)$

\Leftrightarrow esiste un NFA N | $L = L(N)$

\Leftrightarrow esiste un DFA D t.c. $L = L(D)$ \Leftrightarrow

esiste una grammatica G regolare t.c. $L = L(G)$.

Ricordiamo che una grammatica è regolare se è libera (le produzioni sono del tipo $A \rightarrow aB$, o $A \rightarrow \varepsilon$).

Sostanzialmente, abbiamo 4 armi per dimostrare che un certo linguaggio è regolare.

Esercizio

$\{w \mid w$ è una stringa sull'alfabeto $\{a, b\}$, e b occorre un numero dispari di volte $\}$

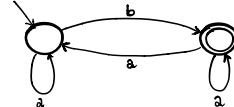
Soluzione: vedi lezione successiva.

Ieri ci siamo lasciati chiedendoci se il seguente linguaggio è regolare.
 $\{w \mid w \text{ è una stringa sull'alfabeto } \{a, b\}, \text{ e } b \text{ occorre un numero dispari di volte}\}$

La risposta è sì:

- Epressione regolare: $r = (ba^*b|a)^*ba^*$

- DFA:



Il precedente DFA è minimo? Come si fa a identificare il DFA minimo?

Epressione regolare $\xrightarrow{\text{Thompson}}$ NFA $\xrightarrow{\text{Subset}}$ DFA $\xrightarrow{?}$ min DFA

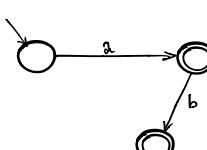
Minimizzazione DFA (light)

Ipotesi fondamentale: il DFA ha funzione di transizione totale. Il perché di questa ipotesi di partenza è dovuto al fatto che l'algoritmo lavora sulla funzione di transizione inversa. Se non è totale, non c'è l'inversa.

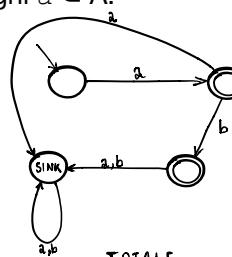
Se il DFA ha funzione di transizione parziale, possiamo trasformarlo in modo tale che la funzione di transizione diventi totale.

1. Aggiungo un nuovo stato, il pozzo (sink).
2. Per ogni stato s di D, se non c'è una transizione uscente per un certo simbolo $a \in A$ (alfabeto), allora aggiungo la transizione etichettata a dallo stato s al pozzo.
3. Aggiungo un self-loop al pozzo, per ogni $a \in A$.

ESEMPIO
 $\{a|ab\}$



FUNZIONE DI TRANSIZIONE PARZIALE



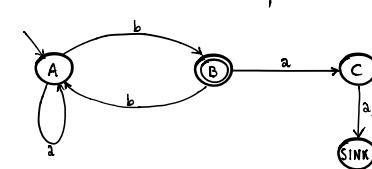
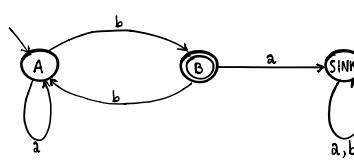
TOTALE

Algoritmo di partition-refinement: vediamolo in azione e intuire come funziona prima di formalizzarlo.



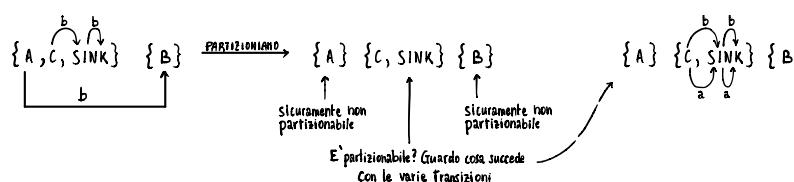
La funzione di transizione è totale? NO.

Due trasformazioni equivalenti:



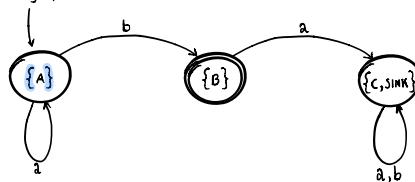
Noi usiamo questa ✓

Partiamo da un partizionamento grossolano: $\{A, C, \text{SINK}\} \quad \{B\}$
 Insieme degli stati non finali

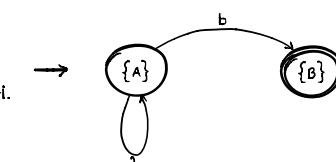


Gruppo/partizione: non è incrementabile, solo spezzabile.
 Sono finali i gruppi che rappresentano stati finali.

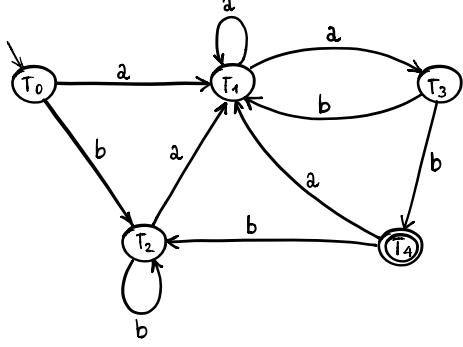
I nodi ora sono i gruppi



Non è ancora minimo: dopo che ho fatto partition refinement, devo eliminare gli stati senza transizione (dead states) e i pozzi.



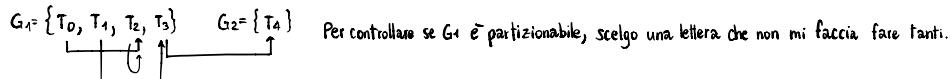
Riprendiamo il DFA di ieri, che riconosce il linguaggio denotato dall'espressione regolare $(a|b)^*abb$:



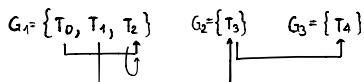
Minimizziamolo. Per cominciare, ci domandiamo se la funzione di transizione è totale: sT → ok.

Procediamo con il raffinamento delle partizioni: $G_1 = \{T_0, T_1, T_2, T_3\}$ $G_2 = \{T_4\}$

Per poter dividere un gruppo, devono esistere dei motivi, delle *ragioni*; trovo delle ragioni per cui non ritengo equivalenti gli elementi del gruppo. Ad esempio, un discriminante per il gruppo G_1 può essere il seguente: da alcuni elementi di G_1 esce una freccia (trasformazione) verso un elemento del gruppo G_2 , da altri esce una freccia verso un elemento di G_1 stesso.

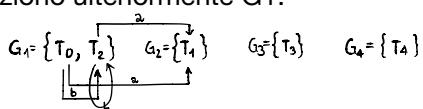


Partizioniamo i due gruppi.

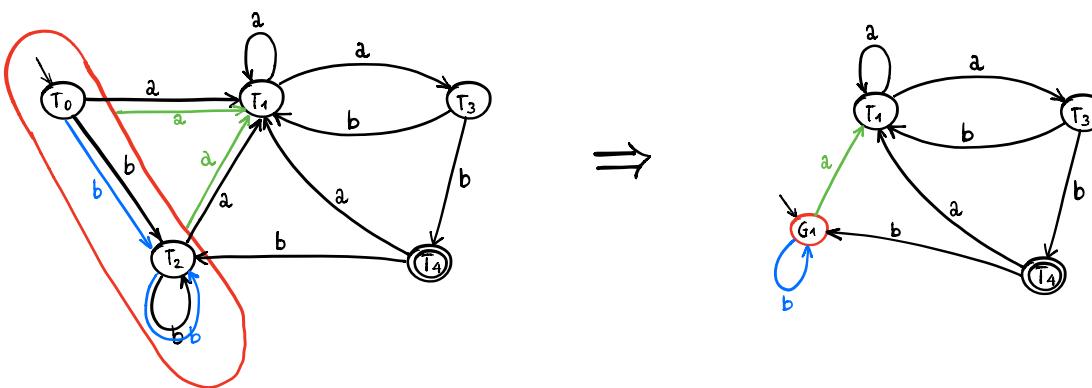


T0, T1 e T2 non sono ancora equivalenti, perché sono differenti nel comportamento delle parole che finiscono sulla b.

Partiziono ulteriormente G1:



• Con le 'a' vado sempre in G2 • Con le 'b' vado sempre in G1



Oggi formalizzeremo ciò che abbiamo visto ieri: la minimizzazione del DFA.

Input: DFA con funzione di transizione totale.

Output: minDFA, cioè quello che ha il numero minimo di stati (nodi) e transizioni (archi), che non necessariamente ha funzione di transizione totale, ed è depurato da dead-states e pozzo.

Questo algoritmo è il più efficiente conosciuto.

Partizionamento: si tratta di effettuare un partizionamento in gruppi, in cui elementi siano equivalenti rispetto ad una nozione specifica di equivalenza.

EQUIVALENZA:

move_d^k è definita per induzione sul valore di k .

$\text{move}_d^0(s, \epsilon) = s \rightarrow$ "Applicata ad un certo stato s , considerando la parola vuota ϵ ".

Ricordiamo che i DFA non prevedono ϵ -transizioni. Qui ϵ è usata solo per dare la definizione induttiva.

$\text{move}_d^{k+1}(s, w) = \text{move}_d(\text{move}_d^k(s, w), a)$ Per muoversi in $k+1$ passi in s rispetto alla parola $w a$.

Sia $D = (S, A, \text{move}_d, s_0, F)$ un DFA con move_d totale. Allora due stati $(s, t) \in S$ sono equivalenti, e si scrive $s \sim t$, se e solo se per ogni parola w di lunghezza k e sull'alfabeto A , la $\text{move}_d^k(s, w) \in F$ se e solo se anche $\text{move}_d^k(t, w) \in F$.
è uno stato finale

Questo esplicita formalmente quello che ieri avevamo trattato intuitivamente: due stati sono equivalenti se, con la stessa parola, riesco a raggiungere uno stato finale (non lo stesso stato finale, ma un certo stato finale).

La minimizzazione del DFA consiste, da un punto di vista tecnico, nel trovare una partizione degli stati del nostro DFA, rispetto a questa nozione di equivalenza. Cioè, vogliamo mettere, nello stesso sottoinsieme di stati di un DFA, tutti quelli che hanno questa caratteristica.

Noi partiamo dal seguente partizionamento grossolano.

$$B_1 = F \quad (\text{stati finali})$$

$$B_2 = S \setminus F \quad (\text{non finali})$$

Questi due blocchi rappresentano l'insieme di quegli stati che si differenziano per parole di lunghezza 0. Cioè, quello che abbiamo è che:

$$\forall s \in B_1, \forall t \in B_2, \quad \begin{aligned} & \text{in quanto: } \cdot \text{move}_d^0(s, \epsilon) = s \in B_1 \subseteq F \quad (\text{finale}) \\ & \cdot \text{move}_d^0(t, \epsilon) = t \in B_2, \text{ quindi certamente } t \notin F. \end{aligned}$$

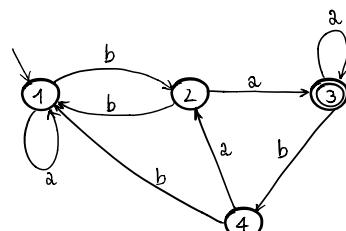
Come facciamo il raffinamento di B_2 ? Quello che dobbiamo ottenere, nel caso in cui riteniamo sia necessario suddividere ancora B_1 e B_2 , sono blocchi che hanno queste caratteristiche:

1. Contengono stati equivalenti,
2. Coppie distinte di blocchi non contengono stati equivalenti (gli stati equivalenti sono obbligati a stare nello stesso blocco).

Cosa facciamo per accertarci di queste due proprietà? Diciamo che:

Data una certa partizione degli stati (es.: nei blocchi $B_1 \dots B_5$), se, per qualche $B_i \neq B_j$ e per qualche $a \in A$, $\exists s, t \in B_i$ t.c. e t.c. $\text{move}_d(s, a) \in B_j$ & $\text{move}_d(t, a) \notin B_j$, allora diciamo che B_i è splittabile rispetto a (B_j, a) .

MENO FORMALMENTE: Se esistono 2 stati s, t di uno stesso blocco B_i , e un simbolo a , tale per cui tramite a arrivo in uno stato B_j per s , ma in uno stato in un blocco $B_j \neq B_i$ per t , allora...



$$B_i = \{1, 4\} \quad B_j = \{2, 3\}$$

s va in B_j , t non va in B_j ma in B_i

B_i può essere splittato rispetto a (B_j, a) .

Cosa è lo split di B_i rispetto a (B_j, a) ?

E' ottenuto rimpiazzando B_i con 2 blocchi dati da $\{s \in B_i \text{ t.c. } move_d(s, a) \in B_j\}$ & $\{s \in B_i \text{ t.c. } move_d(s, a) \notin B_j\}$

Algoritmo di partition refinement

INPUT: DFA totale $D = (S, A, move_d, s_0, F)$

OUTPUT: partizione degli stati di S in blocchi di stati equivalenti secondo la definizione di equivalenza.

$$B_1 = F$$

$$B_2 = S \setminus F$$

$$P = \{B_1, B_2\}$$

while $(\exists B_i, B_j \in P \text{ & } \exists a \in A \text{ con } B_i \text{ splittabile rispetto a } (B_j, a))$

{split di B_i rispetto a (B_j, a) in $P\}$ % ottengo 2 sottoinsiemi:
- stati di B_i che tramite a -transizioni raggiungono stati che stanno in B_j
- stati di B_i che tramite a -transizioni raggiungono stati che stanno in B_i

Algoritmo per la minimizzazione di DFA

INPUT: DFA totale

OUTPUT: min DFA

Let $P = \{B_1, \dots, B_n\}$ il risultato del partition refinement di B .

foreach B_i impostare uno stato temporaneo t_i (poi potremo anche buttarlo via) % con questo foreach decidiamo quali sono gli stati

if $B_i \subseteq F$ (è composto esclusivamente da stati finali) allora t_i è finale % dell'automa che stiamo costruendo

foreach (B_i, B_j, a) t.c. $s_i \in B_i, s_j \in B_j, move_d(s_i, a) = s_j$ % c'è una transizione da s_i a s_j etichettata a

impostare una transizione etichettata a dallo stato temporaneo t_i per B_i allo stato temporaneo t_j per B_j .

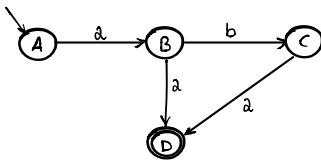
foreach dead temporary state t_i , rimuovere t_i e tutte le transizioni da/a t_i .

• Gli stati del minDFA sono gli stati temporanei rimasti (sopravvissuti) a questa fase qui.

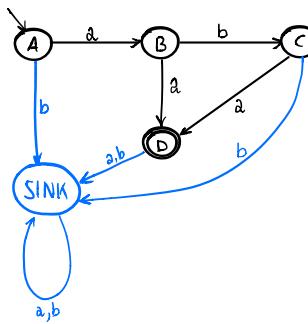
• Le transizioni del minDFA sono le transizioni rimaste.

• Lo stato iniziale del minDFA è lo stato che corrisponde a B_i t.c. $s_0 \in B_i$

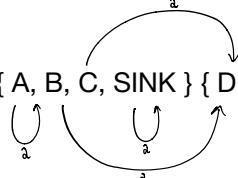
Esercizio 1 - Minimizzare il seguente DFA:



Alfabeto = { a, b }



Non finali = { A, B, C, SINK }, finali = { D }. Riusciamo a spartire il primo blocco? Sì: { A, B, C, SINK } { D }



Otteniamo { A, SINK }, { B, C }, { D }. Riusciamo a spartire qualcosa? Sì: { A, SINK }, { B, C }, { D }

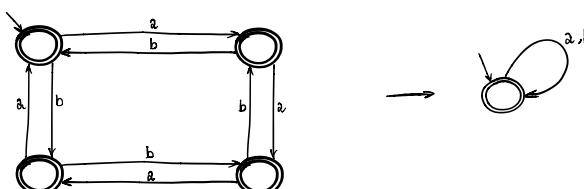


Otteniamo { A, SINK } { B } { C } { D }. Non servirebbe andare oltre, in quanto SINK sarebbe "bruciato". In ogni caso, se si procedesse, ci si accorgerebbe che: { A, SINK } { B } { C } { D } e pertanto il primo blocco verrebbe spezzato.



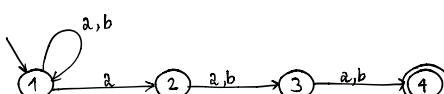
Concludiamo pertanto che il DFA era già minimo.

Esercizio 2 - Minimizzare il seguente DFA:

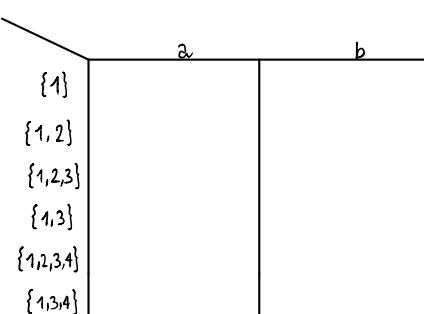
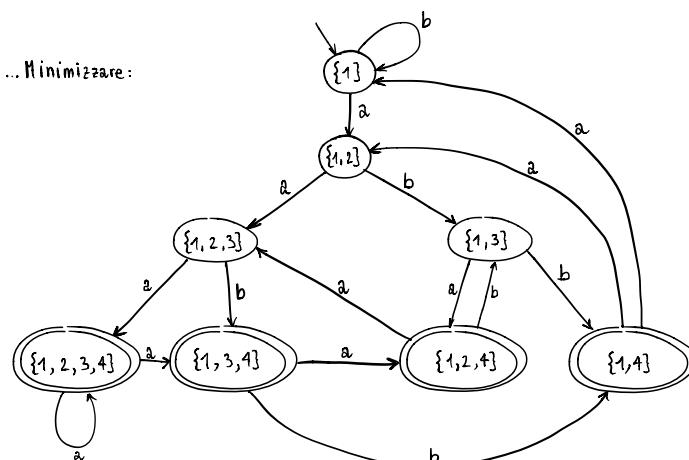


Esercizio 3

Regexp r = (a | b)* a (a | b) (a | b)



... Minimizzare:



STATI FINALI RAPPRESENTANO: (w) aab
(w) aab
(w) aba
(w) abb

Pumping lemma per linguaggi regolari

Sia L un linguaggio regolare. Allora $\exists p \in \mathbb{N}^*$ t.c. $\forall z \in L, |z| > p, \exists u, v, w$ t.c.

($z = uvw$ & $|uv| \leq p$ & $|v| > 0$ & $\forall i \geq 0, uv^i w \in L$) \rightarrow 3 sottostringhe di cui 1 pumpabile

DIMOSTRAZIONE

Se L è regolare, allora -è generato da una grammatica regolare -è riconosciuto da un NFA/DFA -è denotato da un'espressione regolare.

$\exists \text{ DFA } M = (S, A, \text{move}_d, s_0, F)$ t.c. $L(M) = L$

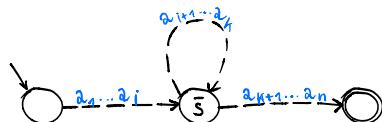
Prendiamo $p = |S|$, il numero di stati di M , che sappiamo esistere.

La massima lunghezza di un cammino in M da s_0 ad uno stato in F che attraversa ogni stato al più una volta è $(p-1)$

Se $z \in L$ è tale che $|z|$ deve essere p , \exists un cammino $z = a_1 \dots a_n$ che attraversa uno stato almeno 2 volte.

Sia \bar{s} uno dei tali stati. Supponiamo che il cammino da s_0 a \bar{s} che non passa attraverso \bar{s} (che quindi lo raggiunge la prima volta) sia etichettato $a_1 \dots a_i$, il cammino da \bar{s} a \bar{s} sia etichettato $a_{i+1} \dots a_K$, che il cammino da \bar{s} allo stato finale sia etichettato $a_{K+1} \dots a_n$.

Allora, ponendo $u = a_1 \dots a_i$, $v = a_{i+1} \dots a_K$, $w = a_{K+1} \dots a_n$, ottengo la tesi.



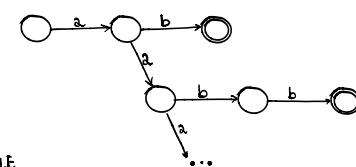
Automa deterministico: 1 solo cammino da stato iniziale a finale.

Ho ipotizzato p stati: cammino lungo $p-1$.

Sto considerando una parola lunga $> p$. Allora c'è (almeno) uno stato da cui passo almeno 2 volte. Supponiamo che tale stato sia quello azzurro.

Il pumping lemma per linguaggi regolari, come al solito, è utile per dimostrare che un linguaggio *non* è regolare.

$\{a^n b^n \mid n \geq 0\}$ senza Pumping lemma...



DIFFICILE

Si usa la negazione: $\nexists p \in \mathbb{N}^*$ t.c. $\exists z \in L, |z| > p, \nexists u, v, w$ t.c. ($z = uvw$ & $|uv| \leq p$ & $|v| > 0$) $\rightarrow \exists i \geq 0, uv^i w \notin L$

Errato fare assunzioni, es. "prendo $p \geq 2$ "

Esercizio

Dimostrare che $\{a^n b^n \mid n \geq 0\}$ è regolare.

uv è confinato qua

Supponiamo che L sia regolare. Sia $p \in \mathbb{N}^*$.

Prendiamo $z = a^p b^p$. $p+p = 2p > p$, z OK.

v ha almeno una 'a'.

Comunque siano uvw , v contiene almeno un'occorrenza di 'a' e contiene solo 'a'

uvw è della forma $a^j b^p$, con $j < p \Rightarrow uv^0 w \notin L$ il che contraddice il pumping lemma.

Proprietà di ϵ -chiusura dei linguaggi regolari

Linguaggi liberi: chiusi per unione; linguaggi regolari: anche.

L sull'alfabeto $A = \{a_1 \dots a_n\}$. $\bar{L} = L((a_1 \dots a_n)) \setminus L$

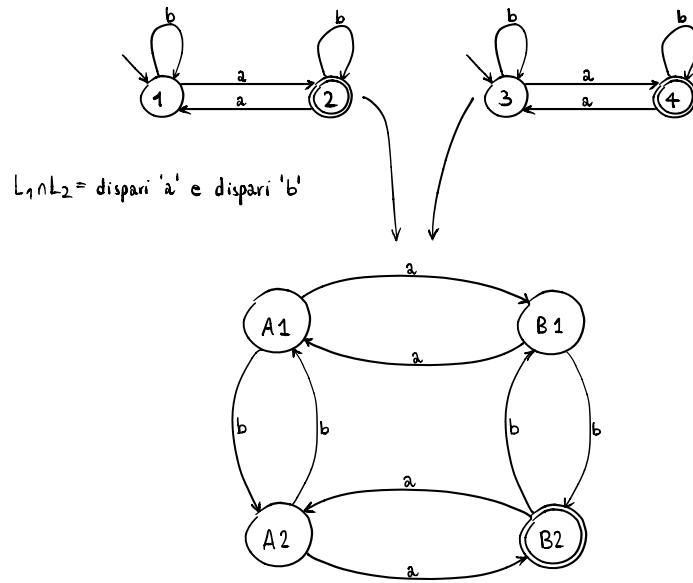
I linguaggi regolari sono chiusi per complementazione.

DIMOSTRAZIONE: Sia L un linguaggio regolare. $\exists \text{ DFA } M$ t.c. $L(M) = L$.

Sia $M^{\text{TOT}} = (S, A, \text{move}_d, s_0, F)$ il DFA con funzione di transizione totale equivalente ad M e sia \bar{M}^{TOT} l'automa $(S, A, \text{move}_d, S \setminus F)$. \bar{M}^{TOT} riconosce $\bar{L} \Rightarrow \bar{L}$ è regolare.

I linguaggi liberi sono chiusi per intersezione? ... sì:

Sia L_1 = linguaggio con dispari occorrenze di 'a', L_2 = linguaggio con dispari occorrenze di 'b'.



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 $M_2 = (S_2, A_2, d_2 := move_{d_2}, s_0, F_2)$.

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

Allora sia $\bar{M} = (\{s_0\}, A_1 \cup A_2, \delta, s_0, \emptyset)$ è t.c. $L(\bar{M}) = L_1 \cap L_2 = \emptyset$.

↑
uno a uno dei due

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

Allora sia $A = A_1 \cup A_2$ e M_{1A} e M_{2A} 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).

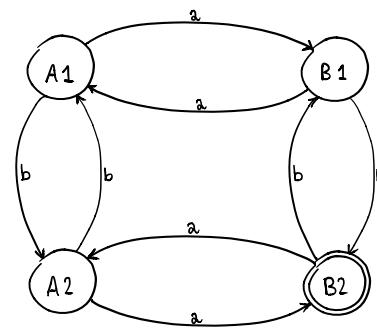
δ è t.c. $\delta((s_i, s_j), a) = (s_i', s_j') \Leftrightarrow S_1^A(s_i, a) = s_i'$ e $S_2^A(s_j, a) = s_j'$. s_0 è lo stato (s_0^1, s_0^2) .

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

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

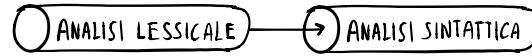
dallo stato iniziale di M_{2A} \exists un cammino a $s_j \Leftrightarrow$

$w \in L_1 \cap L_2$



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...

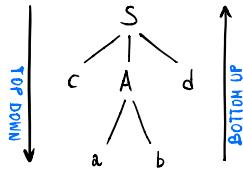


L'analisi sintattica si occupa, in primo luogo, di stabilire se una certa stringa (l'output dell'analisi lessicale) sia in effetti derivabile dalla grammatica del linguaggio.

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.

$$\text{Supponiamo } G : \begin{cases} S \rightarrow cAb \\ A \rightarrow ab|a \end{cases}$$

Un albero di derivazione è



Top-down: costruisce l'albero dalla radice alla frontiera, ed è per questo più facile da capire.

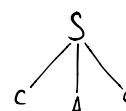
Bottom-up: costruisce l'albero dalla frontiera alla radice (black magic?); la gran parte delle grammatiche dei linguaggi di programmazione sono analizzabili in bottom-up, e pertanto la gran parte degli analizzatori usano questa categoria.

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.

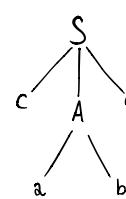
Simulazione

"cad" $\in L(G)$? Devo poter ottenere un albero di derivazione per quella stringa.

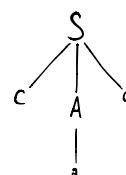
La mia stringa inizia con la 'c': guardo tutte le produzioni della S che iniziano con la 'c'. In questo caso sono fortunato, perché ho solo una produzione che espande la S, e quindi certamente devo usare quella.



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.



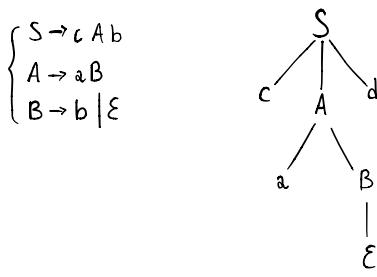
Ho matchato la 'a'; adesso mi aspetto di incontrare una 'd', ma così non è, in quanto in contro una 'b'. Provo ad andare indietro (backtrack), controllando se posso ottenere di meglio.



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:



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.

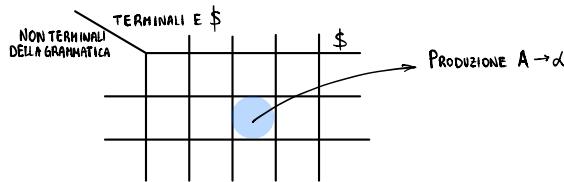
Analizzatore sintattico

Strutture dati: - input buffer - pila - parsing table

Input buffer: parola + terminatore: $w \$$

Pila: inizializzata così: $\boxed{\$}$ la testa della pila contiene la derivazione corrente, 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-termine quando nell'input buffer vediamo un certo simbolo.



ESEMPIO:

$$\begin{cases} S \rightarrow c A b \\ A \rightarrow a B \\ B \rightarrow b \mid \epsilon \end{cases}$$

NON TERMINALI DELLA GRAMMATICA	TERMINALI E \$				
	a	b	c	d	\$
S	ERROR	...	S→cad
A	A→aB
B	...	B→b

Algoritmo:

INPUT: w , parsing table M per G .

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

INITIALIZAZIONE: input buffer $\leftarrow w \$$, pila $\boxed{\$ S}$

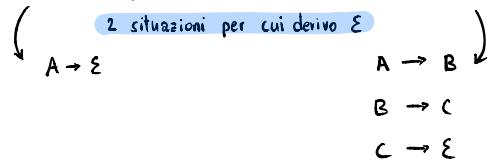
```

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  }
  
```

Per costruire il parser abbiamo bisogno di due concetti, FIRST e FOLLOW.

FIRST(d) = insieme dei terminali che iniziano le stringhe che derivano da d:

$$= \{ a : A \Rightarrow a w_2 \text{ per qualche } w_2 \} \cup \{ \epsilon, \text{ se } A \Rightarrow^* \epsilon \}$$



$$\begin{aligned} G: \quad S &\rightarrow AB \\ &A \rightarrow a|\epsilon \\ &B \rightarrow b \end{aligned}$$

$$L(G) = \{ab, b\}$$

PARSING TABLE:

	a	b	\$
S	$S \Rightarrow AB$	$S \Rightarrow AB$	
A	$A \Rightarrow a$	$A \Rightarrow \epsilon$	
B		$B \Rightarrow b$	

Nella mia tabella devo trovare le direttive che supportano le derivazioni:

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$$

$$S \Rightarrow AB \Rightarrow B \Rightarrow b$$

FIRST: Grammatica $G = (V, T, S, P)$ $S \Rightarrow^* a$

Per ogni simbolo $X \in V$, l'insieme first(X) è calcolato come segue:

1. Se $X = b$ (è un terminale, cioè $X \in T$) allora $\text{first}(X) = \{X\}$

2. $\forall X \in V \setminus T$ (è un non-terminale) inizializzare $\text{first}(X)$ come \emptyset e procedere come segue:

- Se $X \rightarrow \epsilon \in P$ (è una produzione), allora aggiungere ϵ a $\text{first}(X)$.

- Se $X \rightarrow Y_1 \dots Y_n$ con $n \geq 1 \in G$, allora applicare la procedura seguente:

```
j := 1
while (j ≤ n) {
    aggiungere a first(X) ogni b tale che b ∈ first(Y_j)
    if (ε ∈ first(Y_j)) then j = j + 1 else break;
}
if (j = n + 1) then aggiungere ε a first(X).
```

First(d) con d stringa di simboli di V

Sia $d = Y_1 \dots Y_n$ con $n \geq 1$.

Inizializzare $\text{first}(d)$ come \emptyset e applicare la seguente procedura:

```
j := 1
while (j ≤ n) {
    aggiungere a first(d) ogni b tale che b ∈ first(Y_j)
    if (ε ∈ first(Y_j)) then j = j + 1 else break;
}
if (j = n + 1) then aggiungere ε a first(d).
```

first		infine
E → TE'	E	first(T). Se ε ∈ first(T) guarda E'
E' → +TE' ε	E'	+ , ε
T → FT'	T	first(F). Se ε ∈ first(F) guarda T'
T' → *FT' ε	T'	* , ε
F → (E) id	F	id, (

Ordine delle combinazioni ↑

RICORDA:

first: sono terminali o ε: ∀A, first(A) ⊆ Tu[ε]

FOLLOW(A)

$$G = (V, T, S, P)$$

Per calcolarlo, seguire la seguente procedura:

- Per ogni $X \in V \setminus \{S\}$, inizializzare $\text{follow}(X) = \emptyset$

- Inizializzare $\text{follow}(S) = \{\$\}$

- repeat

foreach $A \rightarrow \alpha X \beta \in P$ {

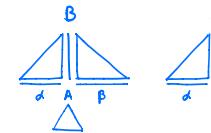
aggiungere $\text{first}(\beta) \setminus \{\epsilon\}$ a $\text{follow}(X)$

if ($\beta = \epsilon$ or $\epsilon \in \text{first}(\beta)$) then

aggiungere $\text{follow}(A)$ a $\text{follow}(X)$

}

until saturation (niente più cambia).



β è ϵ o nullable; produzioni in cui A compare nel body. α, β qualsiasi simbolo di G .

$\beta = \epsilon$ (stringa nulla) o $\text{first}(\beta)$ contiene β

ESECUZIONE

$A \rightarrow \alpha X \beta$	
1	$E \rightarrow T E'$
2	$E \rightarrow + T E' \quad \quad \epsilon$
3	$T \rightarrow F T' \quad \quad \epsilon$
4	$T' \rightarrow * F T' \quad \quad \epsilon$
5	$F \rightarrow (E) \quad \quad \text{id}$

		infine
E	\$ $\text{first}(() \setminus \epsilon = ($	\$)
E'		$\beta = \epsilon$ \$)
T	$\text{first}(E') = +$	\$) + $\beta \in \text{first}(E')$
T'		\$) + $\beta = \epsilon$
F	$\text{first}(T') \setminus \epsilon = *$	\$) + * $\beta \in \text{first}(T')$

Pseudocodice per il calcolo di FOLLOW

Per ogni non-terminale X , inizializza $\text{FOLLOW}(X) = \emptyset$;

$\text{Follow}(S) := \{\$\}$;

Ripeti il ciclo seguente, finché nessun $\text{FOLLOW}(X)$ viene più modificato in una iterazione:

Per ogni produzione $X \rightarrow \alpha Y \beta$

$\text{FOLLOW}(Y) := \text{FOLLOW}(Y) \cup (\text{FIRST}(\beta) - \{\epsilon\})$;

Per ogni produzione $X \rightarrow \alpha Y$ e per ogni produzione $X \rightarrow \alpha Y \beta$ con $\epsilon \in \text{FIRST}(\beta)$

$\text{FOLLOW}(Y) := \text{FOLLOW}(Y) \cup \text{FOLLOW}(X)$;

Vediamo un esercizio

Grammatica:

$E \rightarrow TE'$

$E' \rightarrow \epsilon \mid +E \mid -E$

$T \rightarrow AT'$

$T' \rightarrow \epsilon \mid *T$

$A \rightarrow a \mid b \mid (E)$

FIRST:

$E \quad a, b, ($

$E' \quad \epsilon, +, -$

$T \quad a, b, ($

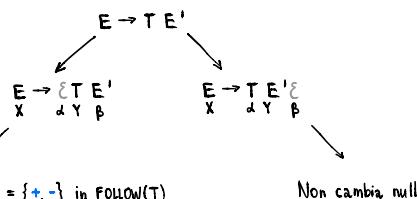
$T' \quad \epsilon, *$

$A \quad a, b, ($

Calcoliamo FOLLOW

Cominciamo identificando le produzioni della forma $X \rightarrow \alpha Y \beta$

• $E \rightarrow TE'$



quindi metto $\text{FIRST}(E') \setminus \epsilon = \{+, -\}$ in $\text{FOLLOW}(T)$

Non cambia nulla.

• $T \rightarrow AT'$



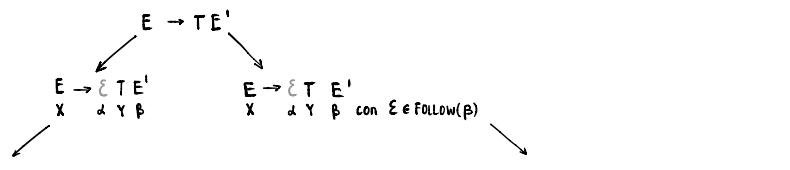
quindi metto $\text{FIRST}(T') \setminus \epsilon = \{*\}$ in $\text{FOLLOW}(A)$

Non cambia nulla.

• $A \rightarrow (E)$ L'unico "mapping" possibile è $A \xrightarrow[X]{=} (E)$ quindi metto $\text{FIRST}() \setminus \epsilon = \{) \}$ in $\text{FOLLOW}(E)$

Ora procediamo identificando le produzioni della forma $X \rightarrow \alpha Y$ e $X \rightarrow \alpha Y \beta$ con $\epsilon \in \text{FIRST}(\beta)$

• $E \rightarrow TE'$



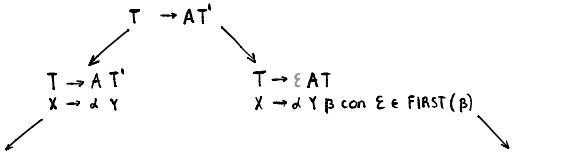
Freccia da $\text{FOLLOW}(E)$ a $\text{FOLLOW}(E')$

Freccia da $\text{FOLLOW}(E)$ a $\text{FOLLOW}(T)$

• $E' \rightarrow +E$ L'unico "mapping" possibile è $E' \xrightarrow[X]{=} +E$ quindi freccia da $\text{FOLLOW}(E')$ a $\text{FOLLOW}(E)$
(L'altro mapping non va bene perché Y deve essere un non-terminale).

• $E' \rightarrow -E$ L'unico "mapping" possibile è $E' \xrightarrow[X]{=} -E$ quindi freccia da $\text{FOLLOW}(E')$ a $\text{FOLLOW}(E)$: già presente
(L'altro mapping non va bene perché Y deve essere un non-terminale).

- $T \rightarrow AT'$



Freccia da FOLLOW(T) a FOLLOW(T')

Freccia da FOLLOW(T) a FOLLOW(A)

- $T \rightarrow *T'$ L'unico "mapping" possibile è $T \rightarrow *T'$ quindi freccia da FOLLOW(T) a FOLLOW(T'): già presente
(L'altro mapping non va bene perché Y deve essere un non-terminale).

RISULTATO

FOLLOW:

E \$)		\$)
E'		\$)
T + -		\$) + -
T'		\$) + -
A *		\$) + - *

Per ogni non-terminale X, inizializza $\text{Follow}(X) = \emptyset$

$\text{Follow}(S) := \{\$\}$

Ripeti il ciclo seguente, finché nessun $\text{Follow}(X)$ viene più modificato in una iterazione:

Per ogni produzione $X \rightarrow \alpha Y \beta$ % Come vedremo dopo, una certa produzione può avere più mapping

$\text{Follow}(Y) := \text{Follow}(Y) \cup (\text{First}(\beta) \setminus \{\epsilon\})$

Se $\beta = \epsilon$ oppure $\epsilon \in \text{First}(\beta)$

$\text{Follow}(Y) := \text{Follow}(Y) \cup \text{Follow}(X);$ % Freccia da $\text{Follow}(X)$ a $\text{Follow}(Y)$

Esercizio 1

Grammatica

$S \rightarrow a A B b$
 $A \rightarrow A c \mid d$
 $B \rightarrow C D$
 $C \rightarrow e \mid \epsilon$
 $D \rightarrow f \mid \epsilon$

<u>FIRST</u>	<u>FOLLOW</u>
$S: a$	$\$$
$A: d$	$e \ f \ b \ c$
$B: e, f, \epsilon$	b
$C: e, \epsilon$	$f \ b \leftarrow$
$D: f, \epsilon$	$b \leftarrow$

CALCOLO DEI FOLLOW

$S \rightarrow a A B b$

Ha due mapping:

$S \rightarrow a \quad A \quad B \quad b$
 $X \rightarrow \alpha \quad Y \quad \beta$
 $\text{Follow}(A) += (\text{First}(Bb) \setminus \{\epsilon\}) = \{e, f, b\}$

$S \rightarrow a \quad A \quad B \quad b$
 $X \rightarrow \alpha \quad Y \quad \beta$
 $\text{Follow}(B) += (\text{First}(\beta) \setminus \{\epsilon\}) = \{b\}$

$A \rightarrow A \ c$

Ha un mapping:

$A \rightarrow \quad A \quad c$
 $X \rightarrow \alpha \quad Y \quad \beta$
 $\text{Follow}(A) += (\text{First}(c) \setminus \{\epsilon\}) = \{c\}$

$A \rightarrow d$

Non ha mapping validi

$B \rightarrow C \ D$

Ha due mapping:

$B \rightarrow \quad C \quad D$
 $X \rightarrow \alpha \quad Y \quad \beta$
 $\text{Follow}(C) += (\text{First}(D) \setminus \{\epsilon\}) = \{f\}$
 $\text{Follow}(C) \leftarrow \text{Follow}(B), \text{ perché } \epsilon \in \text{First}(D)$

$B \rightarrow \quad C \quad D$
 $X \rightarrow \alpha \quad Y \quad \beta$
 $\text{Follow}(D) += (\text{First}(\beta) \setminus \{\epsilon\}): \emptyset$
 $\text{Follow}(D) \leftarrow \text{Follow}(B), \text{ perché } \beta = \epsilon$

$C \rightarrow e$

Non ha mapping validi

$C \rightarrow \epsilon$

Non ha mapping validi

$D \rightarrow f$

Non ha mapping validi

$D \rightarrow \epsilon$

Non ha mapping validi

Esercizio 2

Grammatica

$S \rightarrow aA \mid bBc$

$A \rightarrow Bd \mid Cc$

$B \rightarrow e \mid \epsilon$

$C \rightarrow f \mid \epsilon$

	<u>FIRST</u>	<u>FOLLOW</u>
$S:$	a, b	$\$$
$A:$	e, d, f, c	$\$$
$B:$	e, ϵ	c, d
$C:$	f, ϵ	c

CALCOLO DEI FOLLOW (Omettiamo i mapping invalidi)

$S \rightarrow aA$

Ha un mapping:

$S \rightarrow a A$

$X \rightarrow \alpha Y \beta$

$\text{Follow}(Y) += (\text{First}(\epsilon) \setminus \{\epsilon\}) : \emptyset$

$\text{Follow}(A) \leftarrow \text{Follow}(S)$, perché $\beta = \epsilon$

$S \rightarrow bBc$

Ha un mapping:

$S \rightarrow b B c$

$X \rightarrow \alpha Y \beta$

$\text{Follow}(B) += (\text{First}(c) \setminus \{\epsilon\}) = \{c\}$

$A \rightarrow Bd$

Ha un mapping:

$A \rightarrow B d$

$X \rightarrow \alpha Y \beta$

$\text{Follow}(B) += (\text{First}(d) \setminus \{\epsilon\}) = \{d\}$

$A \rightarrow Cc$

Ha un mapping:

$A \rightarrow C c$

$X \rightarrow \alpha Y \beta$

$\text{Follow}(C) += (\text{First}(c) \setminus \{\epsilon\}) = \{c\}$

Grammatica

$$\begin{aligned} A &\rightarrow aBc \mid bDd \\ B &\rightarrow \epsilon \mid e \\ D &\rightarrow \epsilon \mid f \end{aligned}$$

FIRST

A: a, b
B: e, ϵ
D: f, ϵ

FOLLOW

A: \$
B: c
D: d

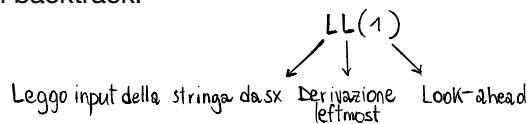
Come riempire la tabella di parsing?

Input: Grammatica G

Output: Tabella di parsing per parsing top-down predittivo

```
foreach (A → α ∈ Produzioni) do
    ∀ b ∈ FIRST(α), inserire A → α nella entry [A, b] della tabella
    if ε ∈ FIRST(α), ∀ x ∈ FOLLOW(A), inserire A → α nella entry [A, x] della tabella
end foreach
inserire error() in tutte le entry vuote della tabella
```

Grammatica LL(1): se la tabella di parsing top-down per G non ha entry multiply-defined, allora la grammatica è LL(1), cioè non farà uso del backtrack.



Look-ahead 1: consulto un solo simbolo “in avanti” per decidere quale produzione utilizzare nell’espansione di ciascun non-terminale. Il look-ahead 1 si visualizza notando che la tabella di parsing ha un solo simbolo sulle colonne.

S → aSb | ab è LL(1)?

a	b	\$
S → aSb		
S → ab		

Entry multiply-defined: non so quale produzione applicare nel momento in cui devo espandere la S vedendo a...\$. Mentre invece, se potessi consultare 2 simboli e leggessi ab...\$, saprei scegliere in maniera opportuna...ma non questa non sarebbe LL(1).

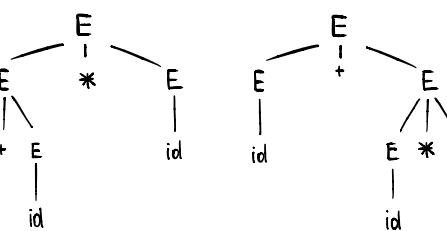
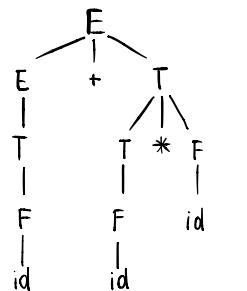
Vediamo ora la seguente grammatica non ambigua, una “queen grammar”:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

Che disambigua la seguente grammatica delle espressioni aritmetiche:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

L’albero di derivazione per la stringa id + id * id usando le due seguenti grammatiche sarebbe:



- queen grammar -

- due alberi della grammatica ambigua -

La "queen grammar" è LL(1)? Per rispondere, dobbiamo controllare che la corrispondente tabella di parsing non contenga entry multiply-defined.

Cominciamo calcolando FIRST e FOLLOW della grammatica:

FIRST:

E: (, id

T: (, id

F: (, id

FOLLOW:

E: \$, +,) \curvearrowleft : \$, +,)
T: * \curvearrowleft : \$, +,), *
F: \curvearrowleft : \$, +,), *

Riempiamo la tabella:

	id	+	*	()
E	$E \rightarrow E + T$				
T					
F					

È analizzabile in top-down? No, ha alcune caratteristiche fastidiose, in particolare la left-recursion: $A \xrightarrow{+} A \downarrow$

La libertà di scelta fa sì che potenzialmente sceglieremo la produzione (?) errata e la complessità diverge.

$E \Rightarrow E + T \Rightarrow E + T + T$

$T \Rightarrow T * F$

Per eliminare la left-recursion, si manipola la grammatica.

Una grammatica $G = (V, S, T, P)$ è **ricorsiva a sinistra** (anche detto: mostra ricorsione a sinistra) se, per qualche $A \in V \setminus T$ e per qualche $a \in V^*$, $A \xrightarrow{*} A \downarrow$ in un certo numero di passi

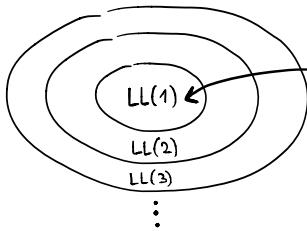
Ad esempio, $\begin{cases} S \Rightarrow B | e \\ B \Rightarrow Sa \end{cases}$

Si dice che la ricorsione a sinistra è **immediata** se, per qualche $A \in V \setminus T$ e per qualche $a \in V^*$, $A \rightarrow Aa \in P$ (è a tutti gli effetti una produzione della grammatica).

comincia proprio con A

$A \xrightarrow{+} A \downarrow$

Stavamo parlando di top-down parsing predittivo.



Dal punto di vista della programmazione, ci si sforza di progettare delle grammatiche di questo tipo.

Esercizio

Grammatica che genera linguaggio delle espressioni aritmetiche

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

First Follow:

$$\begin{array}{ll} E: & (, id, \$, +,) \\ T: & (, id, * \\ F: & (, id \end{array}$$

$$\begin{array}{ccccccc} & id & + & * & (&) & \$ \\ \hline E & E \rightarrow E + T & & & E \rightarrow E + T & & \\ & E \rightarrow T & & & E \rightarrow T & & \\ \hline \hline T & T \rightarrow T * F & & & T \rightarrow T * F & & \\ & T \rightarrow F & & & T \rightarrow F & & \\ \hline \hline F & F \rightarrow id & & & F \rightarrow (E) & & \end{array}$$

Riprendiamo il concetto di **ricorsione a sinistra**.

La grammatica di prima presenta ricorsione immediata a sinistra:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Un'altra grammatica:

$$S \rightarrow Bb$$

$$B \rightarrow Sa$$

Mostra ricorsione in due passi:

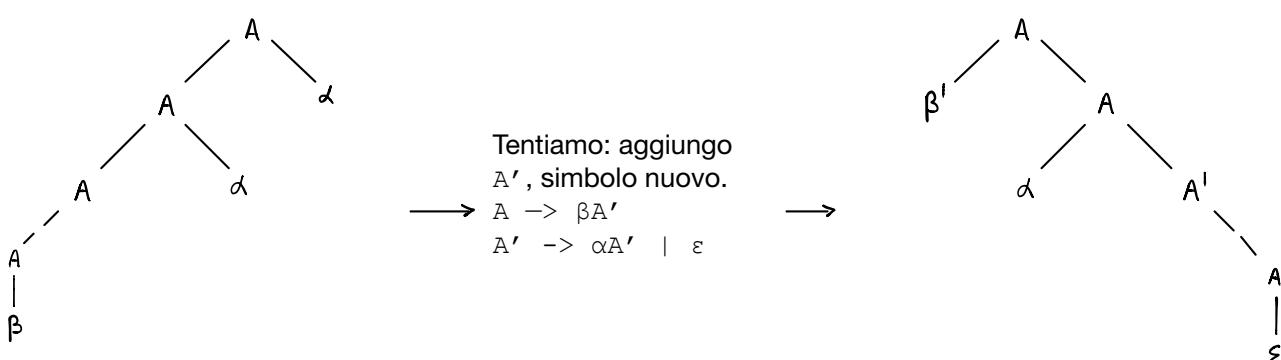
$$S \Rightarrow Bb \Rightarrow Sab$$

$$\text{Ossia, } S \Rightarrow^2 S ab$$

Come modificare una grammatica per eliminarne la ricorsione a sinistra ma mantenendola equivalente?

Focus: vediamo il caso delle grammatiche immediantemente ricorsive a sinistra.

$A \rightarrow A\alpha \mid \beta$ con $\beta \neq Ay$ per qualunque y (gamma), e con $\alpha \neq \varepsilon$



In generale, la ricorsione immediata è in una forma

$$A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_k \quad \text{con } \beta_i \neq A\gamma_i \quad \forall i = 1 \dots k \text{ e } \alpha_j \neq \varepsilon \quad \forall j = 1 \dots n$$

Per eliminare la ricorsione a sinistra:

$$A \rightarrow \beta_1 A' \mid \dots \mid \beta_k A'$$

$$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$$

Focus: Vediamo una grammatica con ricorsione non immediata.

A → Ba | b

B → Bc | Ad | b

- Occupiamoci della produzione B → Bc: la teniamo, non ci dà troppo fastidio perché sappiamo come gestire la ricorsione immediata: B → Bc | **Ba**d | **b**d | b
- La produzione scomoda è B → Ad, che trasformiamo in:

B → bdB' | bB'

B' → cB' | adB' | ε

Esercizio

Eliminare la ricorsione immediata a sinistra e determinare se la grammatica trasformata è LL(1)

E → E + T | T

T → T * F | F

F → (E) | id

Diventa:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

La grammatica trasformata è LL(1) in quanto, se facessimo la tabella di parsing top-down predittivo, non vi sarebbero entry multiply-defined (è in linea con le nostre attese).

Vedremo che:

TOP DOWN: è fastidiosa la ricorsione a sinistra.

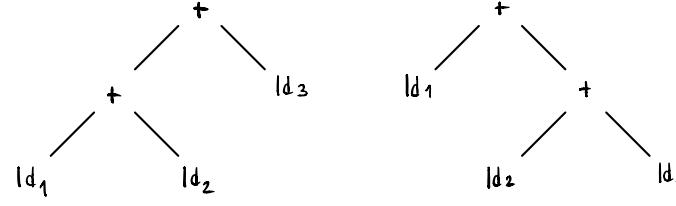
BOTTOM UP: è fastidiosa la ricorsione a destra.

L'eliminazione della ricorsione a sinistra non è una cura rispetto all'ambiguità delle grammatiche.

Per constatare questa affermazione, vediamo un esempio.

La grammatica $E \rightarrow E + E \mid E * E \mid (E) \mid id$ è ambigua in quanto posso proporre una stringa con due alberi di derivazione con medesima frontiera:

$Id_1 + Id_2 + Id_3$



Eliminazione della ricorsione a sinistra:

$A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_k$

Diventa:

$A \rightarrow \beta_1 A' \mid \dots \mid \beta_k A'$

$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$

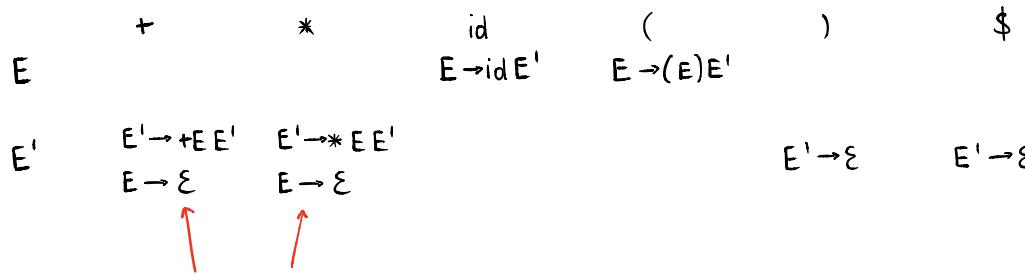
Nella grammatica precedente: $E \rightarrow E \underbrace{+ E}_{d_1} \mid E \underbrace{* E}_{d_2} \mid \underbrace{(E)}_{\beta_1} \mid \underbrace{id}_{\beta_2}$

Seguendo le regole specificate poco prima otteniamo:

$E \rightarrow (E) E' \mid id E'$

$E' \rightarrow +EE' \mid *EE' \mid \varepsilon$

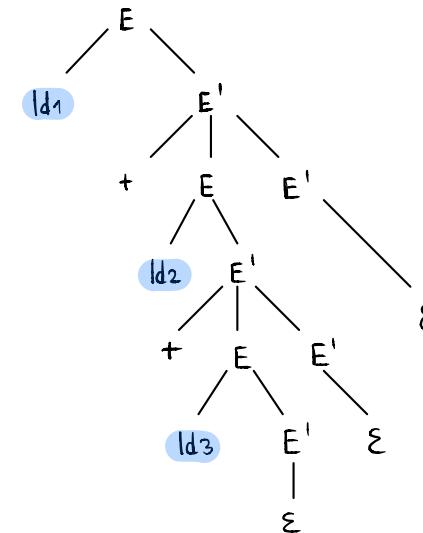
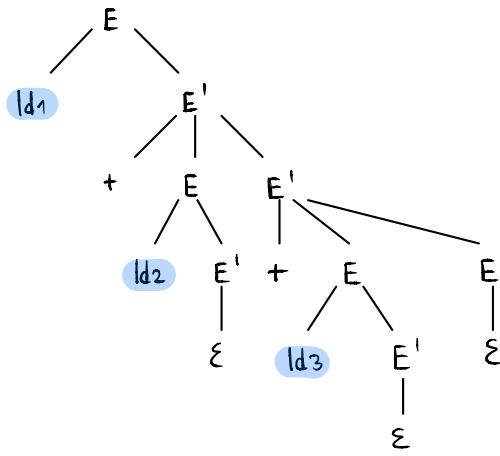
FIRST	FOLLOW
$E \quad (, id$	$\$,), +, *, \varepsilon$
$E' \quad +, *, \varepsilon$	$\$,)$



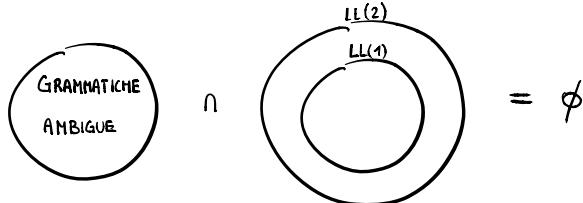
Entry multiply-defined! La grammatica non è LL(1).

Abbiamo constatato che l'eliminazione della ricorsione a sinistra non elimina anche l'ambiguità.

Proviamo ad usare la tabella di parsing per risolvere $Id_1 + Id_2 + Id_3 \$$. Otteniamo due alberi distinti ma con medesima frontiera: ambiguo!



Attenzione quindi alla seguente relazione insiemistica:



Ora, basta parlare di ricorsione a sinistra; parliamo d'altro.

La grammatica $S \rightarrow aSb \mid ab$ è LL(1)? No! Non c'è nemmeno bisogno della tabella. Infatti:

- Ho due produzioni il cui body inizia con lo stesso terminale;
- Per ogni stringa aSb e ab , i FIRST sono entrambi a .

Quindi, nella tabella ho sicuramente la entry

$$S \begin{array}{c} a \\ \hline S \rightarrow aSb \\ S \rightarrow ab \end{array}$$

Questo è il caso di tutte le grammatiche con più di una produzione con gli stessi elementi nei FIRST.

Si parla in questo caso di grammatiche che possono essere **fattorizzate a sinistra**.

La grammatica precedente è un esempio di grammatica che può essere fattorizzata a sinistra, cioè tale che, per qualche A , si hanno produzioni $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$ tali che $\bigcap_{i=1,\dots,n} \text{FIRST}(\alpha_i) \setminus \{\epsilon\} \neq \emptyset$

Si può riscrivere in una grammatica equivalenti che non presenti questo problema?

Caso generale:

Trasformare $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ per togliere il prefisso comune α .

$$A \rightarrow \alpha A'$$

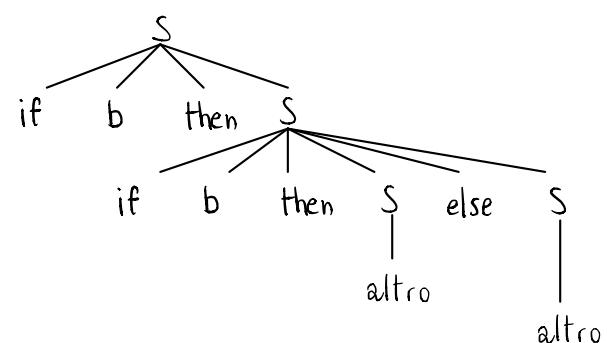
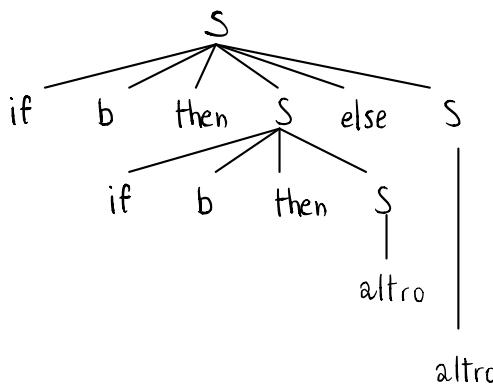
$$A' \rightarrow \beta_1 \mid \beta_2$$

$S \rightarrow aSb \mid ab$ Ritardo la scelta sulla produzione
input buffer: $a \dots$

	FIRST	FOLLOW	a	b	\$
$S \rightarrow aS'$	a	\$b	$S \rightarrow aS'$		
$S' \rightarrow Sb \mid b$	ab	\$b	$S' \rightarrow Sb$	$S' \rightarrow b$	

Dangling Else

La grammatica seguente, detta del *dangling else*, è ambigua: $S \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{altro}$
Infatti, nella stringa *if b then if b then altro else altro* non ci si capisce a cosa si accoppi *else* (se al primo o al secondo *if*).



La grammatica è ambigua, su questo non ci piove. Può essere fattorizzata a sinistra?

$$S \rightarrow \text{if } b \text{ then } S \ S'$$

$$S' \rightarrow \text{else } S \mid \epsilon$$

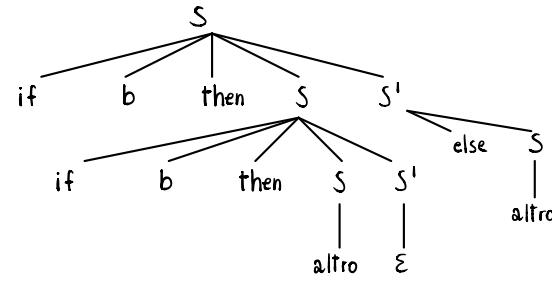
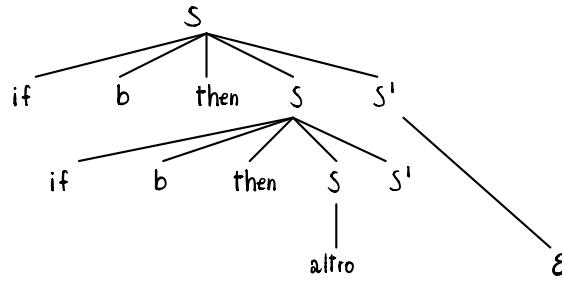
La nuova versione è LL(1)?

$$\begin{array}{ll} \text{FIRST} & \text{FOLLOW} \\ S & \text{if, altro} \\ S' & \text{else, } \epsilon \end{array} \quad \xrightarrow{\text{informazione vuota, ma tecnicamente c'è}}$$

$$S' \begin{array}{c} \text{else} \\ \hline S' \rightarrow \text{else } S \\ S' \rightarrow \epsilon \end{array}$$

NON E' LL(1).

Oltretutto, è ambigua? Sì, ecco un esempio che lo dimostra:



2 possibilità per disinnescare il dangling else:

- Ogni `then` è abbinato ad 1 `else`;
- Si impone l'**innermost binding**: ogni `else` è abbinato al più vicino `then unmatched`: `if b then if b then altro else altro`

La seconda opzione (innermost binding) si può conseguire facendo uso di una grammatica migliore:

$S \rightarrow M \mid U$ ($M = \text{"Matched"}, U = \text{"Unmatched"}$)

$M \rightarrow \text{if } b \text{ then } M \text{ else } M \mid \text{altro}$

$U \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } M \text{ else } U$

Questa grammatica consegna l'innermost binding in quanto, tra un `then` e un `else`, ci sono sempre cose `Matched`, cioè non c'è un `then` disaccoppiato.

Bottom-up parsing: Ricostruzione di un albero di derivazione rightmost, per una data stringa di ingresso:

- Si parte dalla frontiera (foglie) dell'albero,
- Si procede verso la radice dell'albero (etichettata con lo start symbol della grammatica).

Tipologie:
 LALR(1):
 - la nostra
 - YACC, Bison LR(1)

$$S \Rightarrow^* d A w$$

con un certo numero di passi
per convenzione, indica tutti i simboli non-terminali

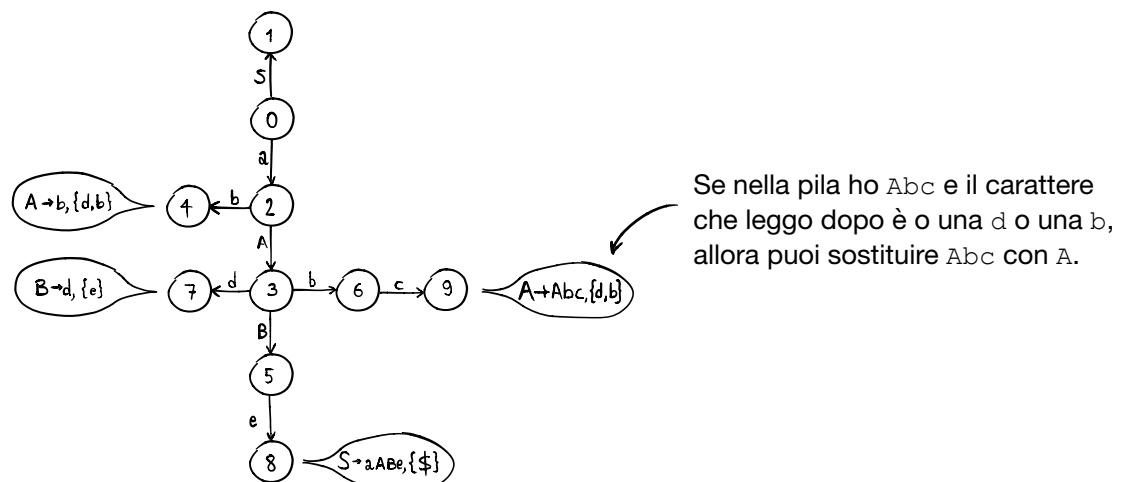
Supponendo che $A \rightarrow \beta \in \text{Produzioni}$, $S \Rightarrow^* d A w \Rightarrow^* d \beta w \Rightarrow^* w \in \text{Linguaggio}$

Le decisioni fondamentali nel parsing bottom-up riguardano *quando* effettuare una riduzione e *quale* produzione applicare, cioè capire quando è il momento di dire che β è l'espansione di A .

Partiamo da piccoli passi, con la seguente grammatica:

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

Da cui deriveremo in seguito l'automa, ma noi lo anticipiamo e lo disegniamo qua sotto:



I cammini dell'automa ripercorrono il driver delle produzioni.

Spiegazione del libro

Il parsing bottom-up è il processo di progressiva *riduzione* di una stringa di ingresso w fino al simbolo iniziale della grammatica.

Prima di vedere un esempio, diamo la definizione di **handle**, o maniglia:

- Un handle è una sottostringa corrispondente al corpo di una produzione.
- La sua derivazione rappresenta un passo nel processo di derivazione destra al rovescio.

Grammatica:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{Id} \end{aligned}$$

Stringa di ingresso:

$\text{Id1} * \text{Id2}$

Un parsing corretto ed efficace è:

Forma sentenziale	Handle	Produzione riducente
-------------------	--------	----------------------

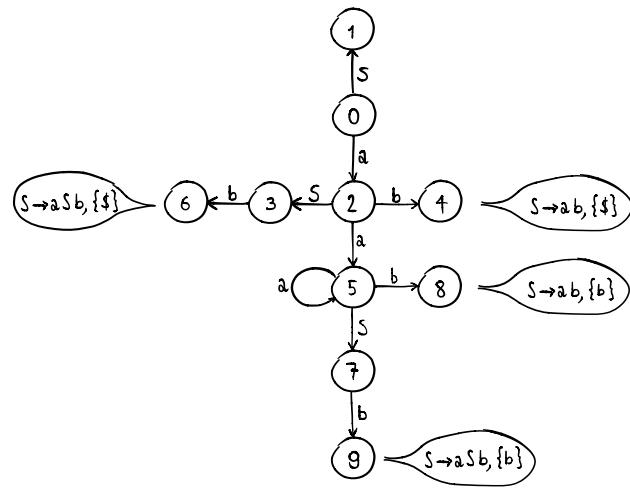
$\text{Id1} * \text{Id2}$	Id1	$F \rightarrow \text{Id}$
$F * \text{Id2}$	F	$T \rightarrow F$
$T * \text{Id2}$	Id2	$F \rightarrow \text{Id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

E //È di fatto una produzione destra: $E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$

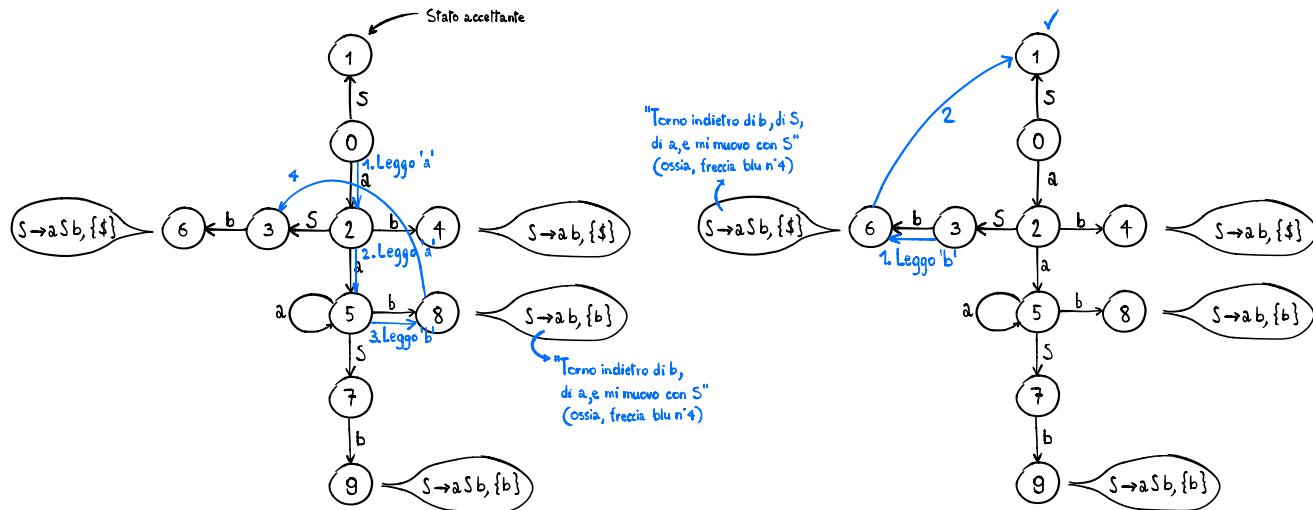
Attenzione che, nonostante la T in grassetto sia corpo della produzione $E \rightarrow T$, non può essere handle per la forma sentenziale $T * \text{Id2}$, altrimenti otterremmo la forma $E * \text{Id2}$ che non può essere derivata dal simbolo iniziale E .

Pertanto si conclude che la sottostringa sinistra corrispondente al corpo di una qualche produzione non deve necessariamente essere un handle.

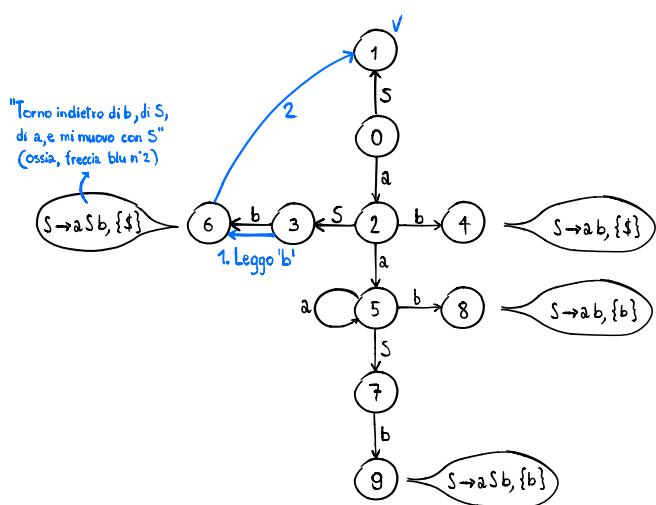
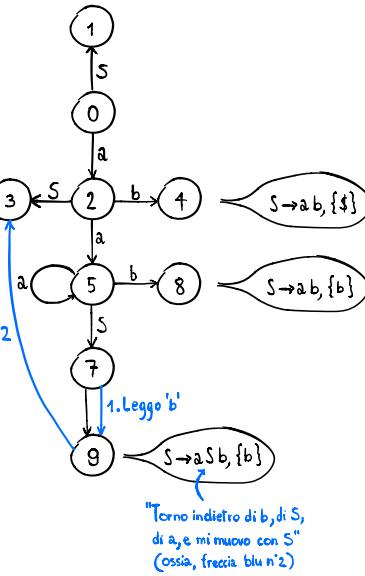
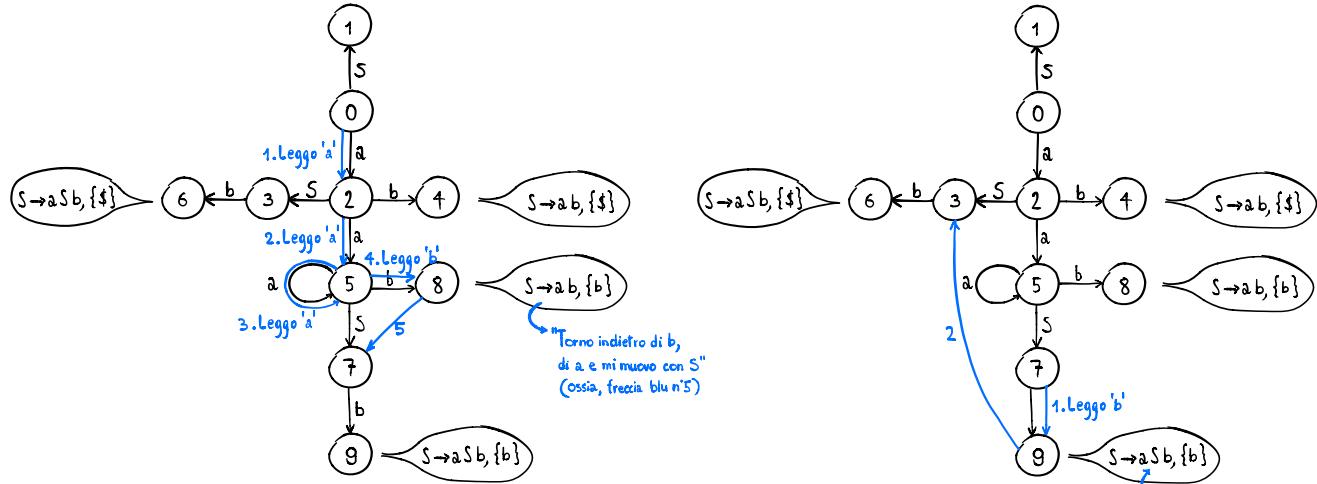
Vediamo la grammatica $S \rightarrow aSb \mid ab$ che ha il seguente automa:



Come si deriva la stringa aabb? [$S \Rightarrow aSb \Rightarrow aabb$]

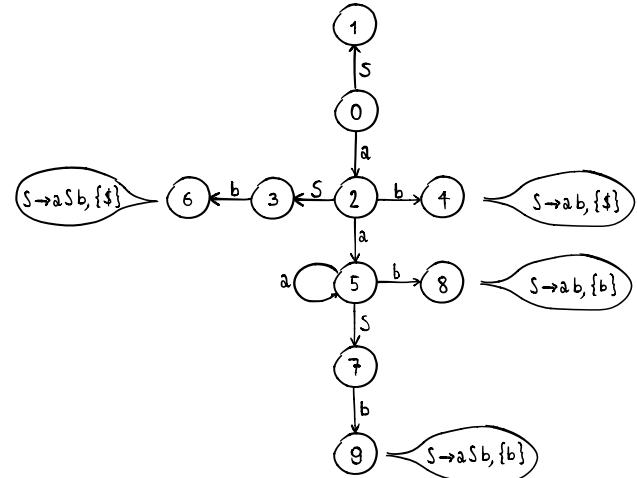


Come si deriva la stringa aaabbb? [S => aSb => aaSbb => aaabbb]



Vediamo la tabella di parsing canonico $LR(1)$ per l'automa visto nella lezione precedente, per la grammatica $S \rightarrow aSb \mid ab$:

	a	b	\$	S
0	shift 2			1
1			accept	1
2	shift 5	shift 4		3
3		shift 6		
4			reduce S->ab	
5	shift 5	shift 8		7
6			reduce S->aSb	
7		shift 9		
8		reduce S -> ab		
9		reduce S -> aSb		



Il parsing **shift-reduce** è una forma di parsing bottom-up

Algorithm 1: Shift-Reduce Algorithm

input : bottom-up parsing table T for \mathcal{G} ; w
output : the rightmost derivation of w in reverse order if $w \in \mathcal{L}(\mathcal{G})$
error() if $w \notin \mathcal{L}(\mathcal{G})$

data structures: $input_buffer$ (init: $w\$$);
 $state_stack$ (init: initial state of the CA underlying T);
 $symbol_stack$ (init: empty)

VARIABLE $b \leftarrow \text{get_next_char}(input_buffer)$;

while $true$ **do**

- if** $T[\text{top}(state_stack), b] = \text{shift } P$ **then**
 - push b onto the $symbol_stack$;
 - push P onto the $state_stack$;
 - $b \leftarrow \text{get_next_char}(input_buffer)$;
- else**
 - if** $T[\text{top}(state_stack), b] = \text{reduce } A \rightarrow \beta$ **then**
 - pop $| \beta |$ times from the $symbol_stack$;
 - push A onto the $symbol_stack$;
 - pop $| \beta |$ times from the $state_stack$;
 - push $T[\text{top}(state_stack), A]$ onto the $state_stack$;
 - print($"A \rightarrow \beta"$);
 - else**
 - if** $T[\text{top}(state_stack), b] = \text{accept}$ **then**
 - break;
 - else**
 - break invoking **error()**;

Esercizio 1

INPUT: $w = a_1 a_2 a_3 b_3 b_2 b_1$, precedente tabella di parsing T

DATA STRUCTURES: `input_buffer = w$ = a1a2a3b3b2b1$`, `state_stack = [0]`, `symbol_stack = []`

					a_2, a_3 diventa S	a_2, S, b_2 diventa S	a_1, S, b_1 diventa S
					5, 8 diventa 7	5, 7, 9 diventa 3	2, 3, 6 diventa 1
VARIABLE	a_1	a_2	a_3	b_3	"S → ab"	"S → aSb"	"S → aSb"
symbol_stack	a_1	a_1, a_2	a_1, a_2, a_3	a_1, a_2, a_3, b_3	a_1, a_2, S	a_1, a_2, S, b_2	a_1, S
state_stack	0	0, 2	0, 2, 5	0, 2, 5, 5	0, 2, 5, 5, 8	0, 2, 5, 7	0, 2, 3, 6
					0, 2, 5, 7, 9	0, 2, 3	0, 1
							$T[1, \$] =$ ACCEPT

Esercizio 2

INPUT: $w = a_1 a_1 a_2 b_2$, tabella τ

DATA STRUCTURES: input-buffer = $w \$ = a_1 a_1 a_2 b_2 \$$, state-stack = [0], symbol-stack = []

VARIABILE	a_1	b_1	a_2	
symbol-stack		a_1	a_1, b_1	$\tau[4, b] = \text{ERROR}$
state-stack	0	0,2	0,2,4	

Anticipazione lezioni successive

Per creare la tabella di parsing LR(1) ci sono due passaggi:

- Determinare l'automa caratteristico.
- Determinare i lookahead set per le riduzioni.

Definizione di **LR(1) item**:

$A \rightarrow \alpha \cdot \beta, \Delta (\Delta \subseteq T \cup \{\$\})$

Esempi di possibili LR(1) item:

$S \rightarrow \cdot aS, \{\$\}$

$S \rightarrow \cdot aSb, \{\$\}$

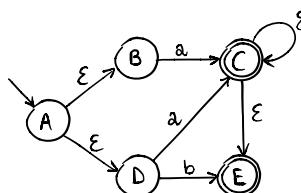
$S \rightarrow \cdot ab, \{\$\}$

Chi sono gli item per $A \rightarrow \varepsilon$?

$A \rightarrow \cdot$

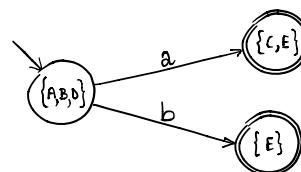
Cominciamo con un ripasso: minimizzazione di un NFA.

NFA:



Troviamo il DFA:

	a	b
{A, B, D}	{C, E}	{E}
{C, E}	⊥	⊥
{E}	⊥	⊥



LR(1)-item:

È una coppia $[l_i, \Delta]$ in cui $\Delta \subseteq (\Sigma \cup \{\$\})$ e l_i è LR(0)-item, cioè una produzione di G con un '.' in una qualche posizione del body.

Vediamo di seguito l'algoritmo per costruire l'automa caratteristico per il parsing LR(1) canonico: prima diamo un'occhiata alla sua struttura.

Struttura

Let $\text{closure}_1(\{[S' \rightarrow \cdot S, \{\$\}]\})$ be the initial state of the automaton, and call it P_0 ;

Tag P_0 as unmarked;

Initialize the collection of states Q to contain P_0 ;

while There is an unmarked state P in Q **do**

 mark P ;

foreach Y such that, for some A, α, β, Δ , the LR(1)-item $[A \rightarrow \alpha \cdot Y \beta, \Delta]$ is in P **do**

 Construct in T_{tmp} the kernel-set of the Y -target of P ;

if Q already contains a state R whose kernel is T_{tmp} then

 Let R be the Y -target of P ;

else

 Let $\text{closure}_1(T_{\text{tmp}})$ be the Y -target of P ;

 Add $\text{closure}_1(T_{\text{tmp}})$ to the collection Q as an unmarked state;

Algoritmo di costruzione dell'automa caratteristico per il parsing LR(1) canonico

$P_0 \leftarrow \text{closure}_1(\{[S' \rightarrow \cdot S, \{\$\}]\})$;

tag P_0 as unmarked;

$Q \leftarrow \{P_0\}$;

while there is an unmarked state P in Q **do**

 mark P ;

foreach grammar symbol Y **do**

$T_{\text{tmp}} \leftarrow \emptyset$;

foreach $[A \rightarrow \alpha \cdot Y \beta, \Delta] \in P$ **do**

 add $[A \rightarrow \alpha Y \cdot \beta, \Delta]$ to T_{tmp} ;

if $T_{\text{tmp}} \neq \emptyset$ **then**

if $T_{\text{tmp}} = \text{kernel}(R)$ for some R in Q **then**

$\tau(P, Y) \leftarrow R$;

else

$\tau(P, Y) \leftarrow \text{closure}_1(T_{\text{tmp}})$;

 add $\tau(P, Y)$ as an unmarked state to Q ;

Definizioni:

- **Item iniziale:** è l'item $[S' \rightarrow \cdot S, \{\$\}]$.

- **Closure item:** questi item hanno il marker all'inizio del body della produzione, ad eccezione di quello iniziale.

- **Kernel item:** tutti gli altri item.

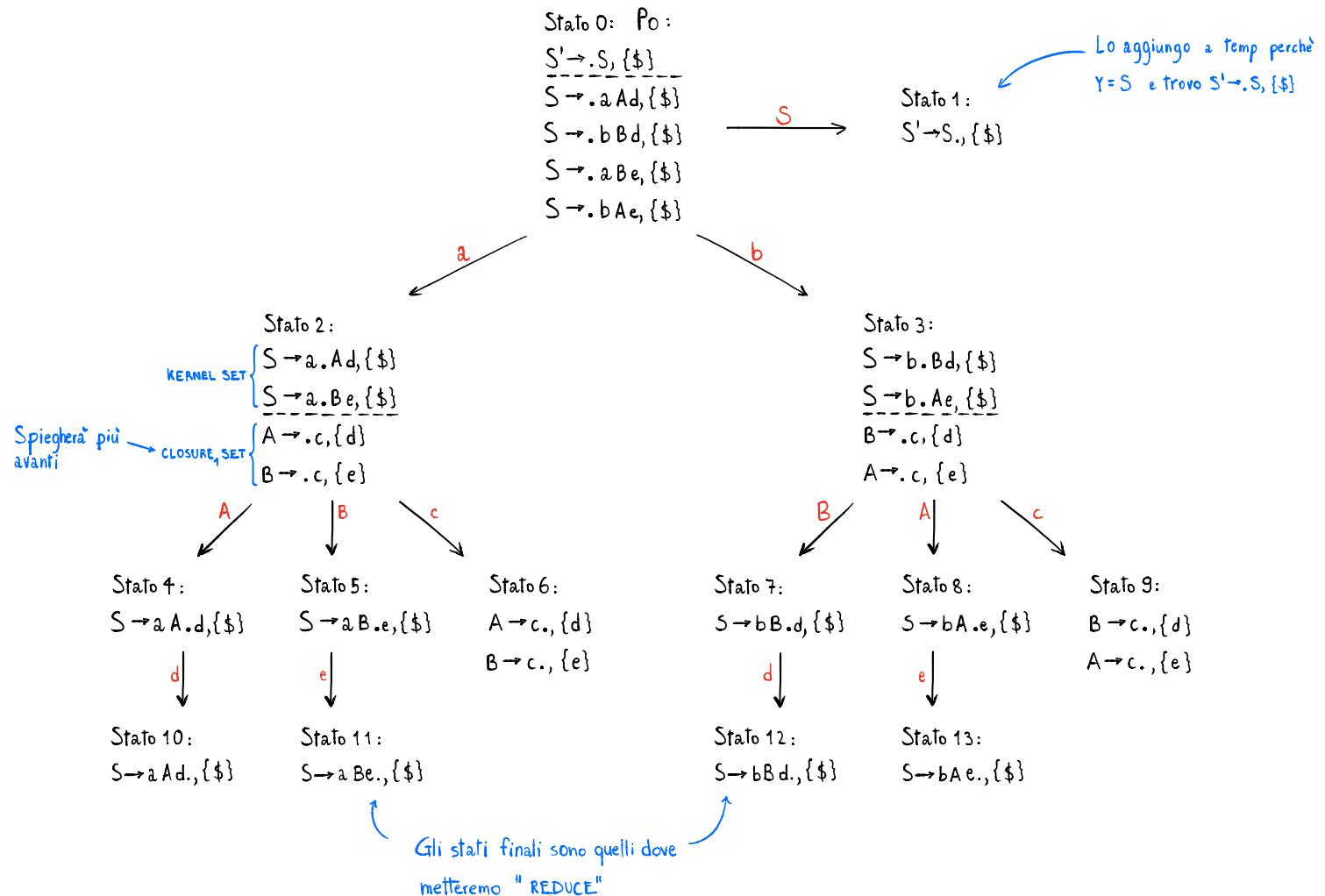
Esempio:

Trovare l'automa caratteristico per la grammatica:

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$



Il canonical LR(1) parsing utilizza l'algoritmo shift-reduce.

L'algoritmo shift-reduce è usato da diverse tecniche di parsing bottom-up. Quello che differenza le varie tecniche è il modo con cui viene costruita la tabella. Nel caso del canonical LR(1) prasing, la tabella viene costruita utilizzando un automa caratteristico che si basa sugli LR(1)-item.

Gli LR item sono coppie $A \rightarrow \alpha.\beta, \Delta$:

- La componente $A \rightarrow \alpha.\beta$ si dice anche LR(0)-item, o semplicemente item. Un *item* è un elemento che contiene almeno una produzione della grammatica, con end-marker denotato da un ‘.’ nel body della produzione.
- La componente Δ è il lookahead-set, che è un sottoinsieme $\Delta \subseteq T^* U \{\$\}$, dove $\$$ è utilizzato come end-marker per le stringhe che voglio analizzare.

Terminologia per gli item

Un item $I = [A \rightarrow \alpha.\beta, \Delta]$ si dice:

- Item Iniziale, se $A \rightarrow \alpha.\beta$ è l'item speciale $S' \rightarrow .S$; ricordiamo che S' è un non-terminale nuovo rispetto alla grammatica data. Questa produzione, $S' \rightarrow .S$, arricchisce la grammatica.
- Ciascun item indica rispetto alla procedura di parsing, che si è già vista/analizzata una certa porzione dell'input buffer, e che questa porzione, rispetto alla produzione indicata dall'item, è tale per cui c'è una parte alfa per la quale è stato tolto il matching con l'input che è stato dato. Quindi questo item è iniziale in quanto essendo il punto immediatamente alla sinistra dello start symbol della grammatica, sta a indicare che ancora non è stato visto nulla della stringa da analizzare.
- Item finale, se la componente $A \rightarrow \alpha.\beta$ è tale per cui il ‘.’ è esattamente alla destra della S , ed indica la situazione opposta: è stato matchato tutto l'input visto rispetto ad una possibile derivazione della S . Serve a capire che siamo arrivati in uno stato.
- Kernel item, se I è iniziale, oppure se α è diverso da epsilon. Pertanto, sono kernel item tutti quelli in cui il ‘.’ non è esattamente all'inizio della produzione, con un'unica eccezione per l'item iniziale, in cui il ‘.’ capita alla sinistra del body della produzione.
- Closure item, o *di chiusura*, se non è kernel item.
- Reducing item, o *di riduzione*, se nella produzione il ‘.’ capita all'estrema destra del body, ossia se $\beta=\epsilon$ oppure se I non è finale.

Ci rimane da dire cos'è l'operazione di closure che applichiamo ad un insieme di item: $\text{closure}_I(\{[S' \rightarrow .S, \{\$\}]\})$.

Nella costruzione dell'automa, nel caso della subset construction, seguivamo questi passi:

“Capivamo come ci muovevamo con un determinato simbolo. Una volta capito ciò, fissavamo l'insieme degli stati possibili dopo aver letto quel simbolo. Una volta fissato questo nucleo di base, calcolavamo l'epsilon closure, cioè andavamo a vedere dove, da questi stati, ci potevamo ancora muovere, utilizzando dei passi di tipo epsilon.”

Quello che facciamo qui è:

Stabilire qual è l'insieme nucleo di item che deve appartenere ad uno stato target rispetto ad una certa transizione dell'automa, e una volta capito quale è questo insieme di item, la chiusura si occupa di dire quali sono gli item che rappresentano delle condizioni equivalenti rispetto alla procedura di parsing. E cioè, supponendo per esempio di partire da un item di questo genere: $A \rightarrow \alpha.B\beta, \Delta$, con un certo lookahead Δ , il minimo che si vuole è dire:

Se io, durante il parsing, mi trovo in uno stato in cui ho già visto la porzione di stringa che corrisponde all'espansione di α , dopo la quale, per poter vedere A , dovrò vedere l'espansione della B e l'espansione della β , allora è equivalente dire, dal punto di vista dell'algoritmo che noi poi eseguiamo per effettuare il parsing, che se la grammatica contiene n produzioni per B che hanno la forma $B \rightarrow \gamma_1, \dots, B \rightarrow \gamma_n$, allora, dopo aver visto l'espansione di α , visto che dobbiamo ancora vedere l'espansione di $B\beta$, potremmo anche trovarci in una qualunque delle posizioni iniziali per queste produzioni della B . Quindi, dimenticando per intanto la Δ , a un insieme di questo tipo, uno ci aggiunge comunque tutti gli item che sono *di chiusura* per B , cioè che hanno il punto esattamente all'inizio del body delle produzioni della B : $B \rightarrow .\gamma_1, \Delta_1; \dots; B \rightarrow .\gamma_n, \Delta_n$, per qualche $\Delta_1, \dots, \Delta_n$.

Naturalmente, qui la ricorsione è regina: chi sono questi $\gamma_1 \dots \gamma_n$? Per esempio, potrebbe essere che γ_1 è una cosa scritta così: $\gamma_1 = c\delta_1$. Ma allora, se io sono all'inizio di δ_1 e se δ_1 comincia con c , è come dire che una delle possibilità è che io veda l'espansione della c seguita dall'espansione di δ_1 , e allora non basta l'aver aggiunto questo, ma bisogna comunque aggiungere tutto ciò che risulta equivalente rispetto all'algoritmo che dobbiamo eseguire poi per lo shift reduce, per tutte e quante le produzioni che sono state elencate all'interno dell'insieme che stiamo costruendo, e in cui capita che ci sia il ‘.’ esattamente davanti ai non-terminali (perché noi sappiamo che, per ogni non-terminale, c'è una produzione).

Quindi uno continua a mettere questo produzioni all'interno dell'insieme; in questo caso specifico, supponendo che per c ci sia la produzione $c \rightarrow a$, uno metterebbe anche $c \rightarrow .a$, con un qualche lookahead Δ_m . E così via per tutti quei casi in cui uno ha inserito una produzione che ha un ‘.’ davanti a un non-terminale.

Se invece il ‘.’ è davanti un terminale, la chiusura non deve aggiungere niente, perché quello che uno aspetta di vedere è esattamente quel terminale nell'input-buffer, che non può essere travestito da altro: se è il terminale c , niente lo può cambiare.

Ora la prof spiega (non molto chiaramente) come sistemare i lookahead set; ometto questa parte. Andiamo direttamente agli esercizi.

Esempio

Grammatica:

$$S \rightarrow aAd \mid cAb$$

$$A \rightarrow e$$

Calcoliamo l'automa caratteristico, considerando che, per quanto riguarda la closure, non ne abbiamo ancora dato una definizione formale, ma ne abbiamo più o meno l'idea del funzionamento.

Stato iniziale:

Lo stato iniziale dell'automa caratteristico ha un kernel set con l'item $S' \rightarrow .S, \{\$\}$, in cui $S' \rightarrow .S$ è una produzione che arricchisce la grammatica.

Dobbiamo ora computare il closure set del kernel set.

Un item che vogliamo chiudere ha la forma $A \rightarrow \alpha.B\beta$. Il lookahead di un item prodotto dalla chiusura è $\Delta_1 = \bigcup_{d \in \Delta} \text{FIRST}(Bd)$

Dove β sono le stringhe che seguono B .

Siccome qui $\beta = \epsilon$ e Δ ha solo un elemento, il $\$$, è molto facile e abbiamo che $\Delta_1 = \text{FIRST}(\$) = \$$.

Pertanto, il closure set include gli item $S \rightarrow .aAd, \{\$\}$ e $S \rightarrow .cAb, \{\$\}$.

Ora ci chiediamo se abbiamo aggiunto item che devono essere chiusi, ma non è questo il caso, in quanto il punto sta, per ognuno dei due item raggiunti, davanti a un non-terminale. Quindi, questo è lo stato iniziale:

$$\begin{aligned} S' &\rightarrow .S, \{\$\} \quad \text{KERNEL SET} \\ \hline - &- - - - - \\ S &\rightarrow .aAd, \{\$\} \quad \text{CLOSURE SET} \\ S &\rightarrow .cAb, \{\$\} \end{aligned}$$

Ogni stato ha tante transizioni quanti sono gli elementi del vocabolario che capitano esattamente alla destra del $'.'$.

Quindi, per il primo stato, ho tre transizioni: una per a , una per c e una per e .

Per costruire lo stato di una transizione, si prendendo tutti e quanti gli item che hanno un $'.'$ davanti all'elemento per cui si sta costruendo la transizione, spostandolo da davanti a dietro.

Stato 1:

Dallo stato iniziale mi muovo con una S -transizione nello stato 1 che contiene nel kernel set l'item $S \rightarrow S., \{\$\}$.

La chiusura non ha niente da aggiungere, in quanto il $'.'$ non sta alla sinistra di un non-terminale. Quindi, lo Stato 1 è:

$$S' \rightarrow S., \{\$\}$$

Stato 2:

Dallo stato iniziale mi muovo con una a -transizione nello stato 2 che contiene nel kernel set l'item $S \rightarrow a.Ad, \{\$\}$.

Dal momento che il punto sta davanti ad A , un un non-terminale, dobbiamo aggiungere item alla chiusura.

La grammatica ha una sola produzione per la A che è $A \rightarrow e$.

Il closure set è quindi costituito dall'item $A \rightarrow .e, \{d\}$ dove d è il FIRST di ciò che segue la A , quindi $\text{FIRST}(d\$) = d$.

La chiusura non ha niente da aggiungere, in quanto il $'.'$ non sta alla sinistra di un non-terminale. Quindi, lo stato 2 è:

$$\begin{aligned} S &\rightarrow a.Ad, \{\$\} \\ \hline - &- - - - - \\ A &\rightarrow .e, \{d\} \end{aligned}$$

Stato 3:

Dallo stato iniziale mi muovo con una c -transizione nello stato 3 che contiene nel kernel set l'item $S \rightarrow c.Ab, \{\$\}$.

$$\begin{aligned} S &\rightarrow c.Ab, \{\$\} \\ \hline - &- - - - - \\ A &\rightarrow .e, \{b\} \end{aligned}$$

Dallo stato 2 abbiamo due transizioni: una per la A , una per la e .

Stato 4:

$$S \rightarrow aA.d, \{\$\}$$

Stato 5:

$$A \rightarrow e., \{d\}$$

Dallo stato 3 abbiamo due transizioni: una per la A , una per la e .

Stato 6:

$$S \rightarrow cA.b, \{\$\}$$

Stato 7:

$$A \rightarrow e., \{b\} // Va bene, non lo ho già: ha un lookahead diverso da quello dell'item nello stato 5.$$

Dallo stato 4 abbiamo una a -transizione verso lo stato 8.

Stato 8:

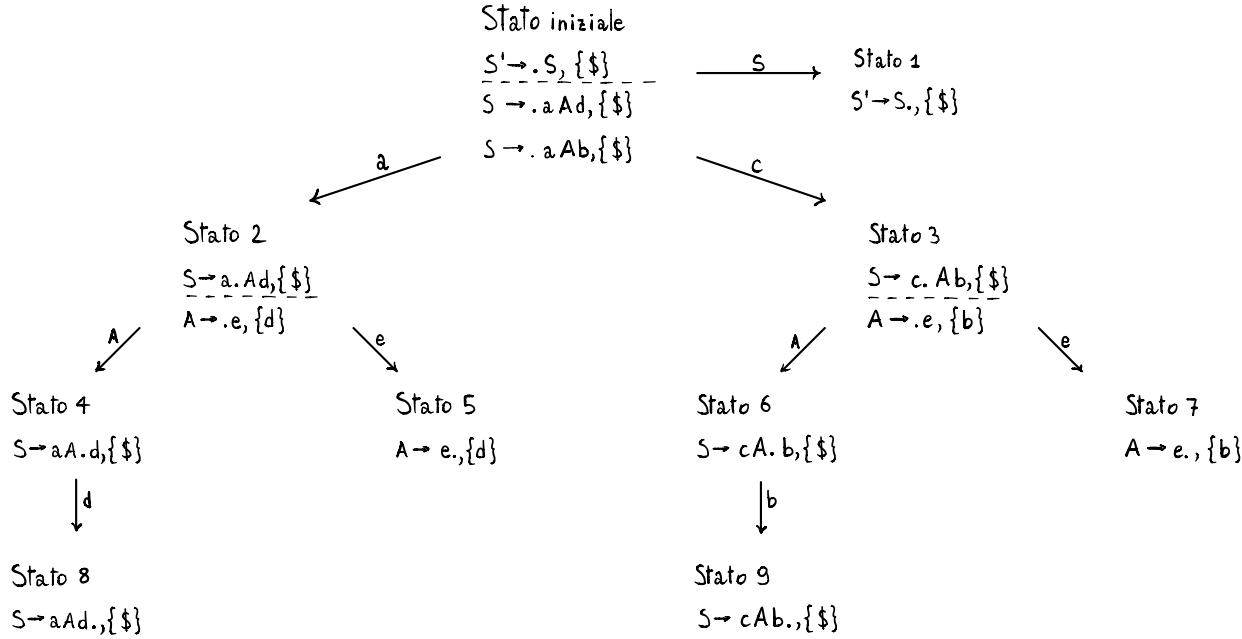
$$S \rightarrow aAd., \{ \$ \}$$

Dallo stato 6 abbiamo una b transizione verso lo stato 9.

Stato 9:

$$S \rightarrow cAb., \{ \$ \}$$

Ora abbiamo finito.



Vediamo ora l'**algoritmo per il calcolo della chiusura di un set P di LR(1)-item**.

function closure1(P)

 tag every item in P as unmarked;

while there is an unmarked item I in P do

 mark I ;

if I has the form $[A \rightarrow \alpha.B\beta, \Delta]$ **then**

$\Delta_1 \leftarrow \bigcup_{d \in \Delta} \text{FIRST}(\beta d); // \Delta_1 = \text{FIRST}(\beta d_1) \cup \dots \cup \text{FIRST}(\beta d_n).$

foreach $B \rightarrow \gamma \in G$ do

if $B \rightarrow \gamma \notin \text{prj}(P)$ **then** // Se ancora non c'è un item che ha $B \rightarrow \gamma$ come prima componente

 add $[B \rightarrow \cdot\gamma, \Delta_1]$ as an unmarked item to P ;

else

if $[B \rightarrow \cdot\gamma, \Gamma] \in P$ and $\Delta_1 \not\subseteq \Gamma$ **then** // Se mi sto dimenticando qualche cosa

 update $[B \rightarrow \cdot\gamma, \Gamma]$ to $[B \rightarrow \cdot\gamma, \Gamma \cup \Delta_1]$ in P ;

 tag $[B \rightarrow \cdot\gamma, \Gamma \cup \Delta_1]$ as unmarked;

 return P ;

L'**if** in azzurro corrisponde alla situazione che abbiamo visto prima nello stato 2.

Avevamo la grammatica $S \rightarrow aAd \mid cAb$, $A \rightarrow e$ e dovevamo chiudere $S \rightarrow aAd, \{ \$ \}$.

Dobbiamo quindi controllare che per la produzione $A \rightarrow e$, dalla forma $B \rightarrow \gamma$,

non vi sia in $\text{prj}(P)$ un item la cui prima componente abbia la forma $B \rightarrow \cdot\gamma$, qui $A \rightarrow \cdot e$.

Questa condizione era verificata, e quindi avevamo aggiunto l'item $A \rightarrow \cdot e, \{ d \}$.

L'**else** in azzurro corrisponde alla seguente situazione.

Ho la grammatica $E \rightarrow E+E \mid E^*E \mid (E) \mid id$.

Lo stato iniziale ha, come kernel item, $E' \rightarrow \cdot E, \{ \$ \}$.

Siccome nel set non ho un item che abbia la prima componente di chiusura per la E , ci metto

$$\begin{aligned} E &\rightarrow \cdot E+E, \{ \$ \} \\ E &\rightarrow \cdot E^*E, \{ \$ \} \\ E &\rightarrow \cdot (E), \{ \$ \} \\ E &\rightarrow \cdot id, \{ \$ \} \end{aligned}$$

Sembra un insieme saturato? No! Noto che vi sono vari non-terminali preceduti dal ' \cdot ' davanti.

Pertanto:

In $E \rightarrow .E + E, \{\$\}$ devo chiudere la E in grassetto portandomi avanti la nozione di chi è il FIRST di $+E, \{\$\}$.

In $E \rightarrow .E * E, \{\$\}$ devo chiudere la E portandomi avanti la nozione di chi è il FIRST di $*E, \{\$\}$.

Mentre gli item $E \rightarrow .(E), \{\$\}$ e $E \rightarrow .id, \{\$\}$ sono a posto.

Questa cosa si cattura con l'**else**:

update $[B \rightarrow .Y, \Gamma]$ to $[B \rightarrow .Y, \Gamma \cup \Delta_1]$ in \mathbb{P} ;

Esempio

Ripetiamo pian piano l'applicazione dell'algoritmo per la grammatica $E \rightarrow E+E \mid E*E \mid (E) \mid id$.

Parto dall'item $E' \rightarrow .E, \{\$\}$.

Quando calcolo Δ_1 , tecnicamente dovrei fare $\bigcup_{d \in \{\$\}} FIRST(d)$, cioè $\Delta_1 = \{\$\}$.

Segue ora il foreach. Vengono aggiunti:

$E \rightarrow .E+E, \{\$\}$ nm (non marcato)
 $E \rightarrow .E*E, \{\$\}$ nm
 $E \rightarrow .(E), \{\$\}$ nm
 $E \rightarrow .id, \{\$\}$ nm

Riparto dal while

Quando considero l'item $E \rightarrow .E+E, \{\$\}$, chi è Δ_1 ? È il FIRST della stringa $+E\$$, cioè $+$. Quindi, $\Delta_1 = \{+\}$.

Entro nell'else. L'else dice che, per tutti gli item che sono di chiusura per la E , venga aggiunto al lookahead set il Δ_1 , e cioè $+$.

Ricordo che *item di chiusura per la E* vuol dire che è tale da avere il $'.'$ davanti al body di una qualunque delle produzioni per la E .

Pertanto, aggiorno gli item a:

$E \rightarrow .E+E, \{\$, +\}$
 $E \rightarrow .E*E, \{\$, +\}$ nm
 $E \rightarrow .(E), \{\$, +\}$ nm
 $E \rightarrow .id, \{\$, +\}$ nm

Riparto dal while

Quando considero l'item $E \rightarrow .E*E, \{\$, +\}$, chi è Δ_1 ? Devo andare a guardare i FIRST di due stringhe: $*E\$$ e $*E+$.

I FIRST sono $*$, quindi $\Delta_1 = \{*\}$.

Entro nell'else, ed aggiorno gli item a:

$E \rightarrow .E+E, \{\$, +, *\}$
 $E \rightarrow .E*E, \{\$, +, *\}$
 $E \rightarrow .(E), \{\$, +, *\}$ nm
 $E \rightarrow .id, \{\$, +, *\}$ nm

Quando riporto dal while per gli item $E \rightarrow .(E), \{\$, +, *\}$ e $E \rightarrow .id, \{\$, +, *\}$, non devo fare niente.

L'insieme finale è costituito da stati marcati:

$E \rightarrow .E, \{E\}$
- - - - - - - -
 $E \rightarrow .E+E, \{\$, +, *\}$
 $E \rightarrow .E*E, \{\$, +, *\}$
 $E \rightarrow .(E), \{\$, +, *\}$
 $E \rightarrow .id, \{\$, +, *\}$

Vediamo ora come si fa, dato un automa caratteristico di questo tipo, a riempire la tabella di parsing.

Tabella di parsing canonical LR(1)

Q: insieme degli stati dell'automa caratteristico.

τ : funzione di transizione dell'automa caratteristico.

La tabella è una matrice di dimensione $|Q| \cdot |V \cup \{\$\}|$.

$\forall Y \in V$ (vocabolario della grammatica):

- Se Y è un terminale e $\tau(P, Y) = R$, allora inserire "shift R" nella entry (P, Y) .
- Se P contiene il reducing item $[A \rightarrow B, \Delta]$, allora inserire "reduce A \rightarrow B" nella entry (P, Y) tali che $Y \in \Delta$.
- Se P contiene l'item finale, allora inserire "accept" in $(P, \$)$.
- Se Y è un non-terminale e $\tau(P, Y) = R$ allora inserire "goto R" (o anche semplicemente "R") nella entry (P, Y) .

Esercizio 1

Grammatica: $S \rightarrow aSb \mid \epsilon$

Costruiamo la collezione degli stati che servono all'automa caratteristico.

Stato iniziale 0:

```
S' → .S, {$}  
/* Devo chiudere la S */  
- - - - -  
S → .aSb, {S} Il lookahead set è dai FIRST di tutto ciò che può seguire S nell'item che mi ha costretto a inserire questi due item.  
S → ., {$} /* Corrisponde alla produzione S → ε */  
/* La chiusura non aggiunge nient'altro, in quanto non ho non-terminali proceduti dal punto */
```

Per lo stato 0 ci aspettiamo due transizioni: una per la S , e una per la a .

Dallo stato 0, con la S , otteniamo un kernel item che prima non avevamo collezionato. Perchè devo preoccuparmi di controllarlo?

Quindi, nuovo stato 1:

```
S' → S., {$}  
/* La chiusura non aggiunge niente */
```

Perchè la chiusura di due kernel uguali non può restituire insiemi diversi! ☺

Dallo stato 0, con la a , otteniamo un kernel item che prima non avevamo collezionato.

Quindi, nuovo stato 2:

```
S → a.Sb, {$}  
/* Devo chiudere la S */  
- - - - -  
S → .aSb, {b} LOOKAHEAD = FIRST(b$)  
S → ., {b}  
/* La chiusura non aggiunge niente */
```

Lo stato 1 e lo stato 2 sono stati completamente processati. Tocca allo stato 2, per il quale ci aspettiamo due transizioni: una per la S , e una per la a .

Dallo stato 2, con la S , otteniamo un kernel item che prima non avevamo collezionato.

Quindi, nuovo stato 3:

```
S → aS.b, {$}  
/* La chiusura non aggiunge niente */
```

Dallo stato 2, con la a , otteniamo un kernel item che prima non avevamo collezionato.

Quindi, nuovo stato 4:

```
S → a.Sb, {b}  
/* Devo chiudere la S */  
- - - - -  
S → .aSb, {b}  
S → ., {b}  
/* La chiusura non aggiunge niente */
```

Lo stato 2 è stato completamente processato. Tocca allo stato 3, per il quale ci aspettiamo una transizione rispetto alla b .

Dallo stato 3, con la b , otteniamo un kernel item che prima non avevamo collezionato.

Quindi, nuovo stato 5:

```
S → aSb., {$}
```

Lo stato 3 è stato completamente processato. Tocca allo stato 4, per il quale ci aspettiamo due transizioni: una per la S , una per la a .

Dallo stato 4, con la S , otteniamo un kernel item che prima non avevamo collezionato.

Quindi, nuovo stato 6:

```
S → aS.b, {b}  
/* La chiusura non aggiunge niente */
```

Dallo stato 4, con la a , otteniamo il kernel item $S \rightarrow aS.b, {b}$, che già abbiamo nello stato 4.

Questo vuol dire che dallo stato 4, con la a , andiamo allo stato 4.

Lo stato 4 e lo stato 5 sono stati completamente processati. Tocca allo stato 6, per il quale ci aspettiamo una transizione rispetto alla b.

Dallo stato 6, con la b, otteniamo un kernel item che prima non avevamo collezionato.

Quindi, nuovo **stato 7**:

$S \rightarrow aSb., \{b\}$

/* La chiusura non aggiunge niente */

Lo stato 6 e lo stato 7 sono stati completamente processati.

Ho finito, passo a disegnare l'automa caratteristico. Dal momento che, successivamente, per riempire la tabella di parsing, ci servirà, oltre all'automa caratteristico, sapere anche quali sono i reducing items e i relativi lookahead set, evidenziamo nell'automa caratteristico i reducing item.

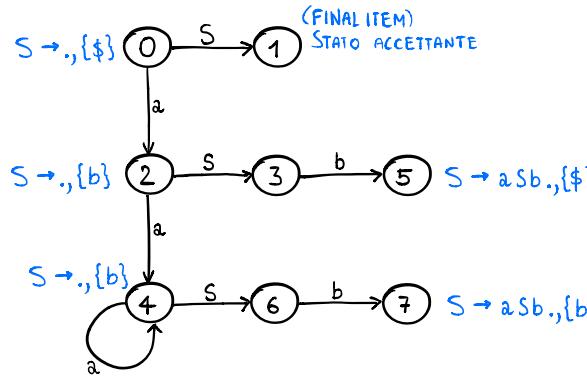
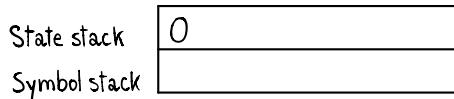


Tabella di parsing: come si riempie?

	a	b	\$	S
0	S2		r "S->ε"	1
1			ACC	
2	S4	r "S->ε"		3
3		S5		
4	S4	r "S->ε"		6
5			r "S->aSb"	
6		S7		
7			r "S->aSb"	

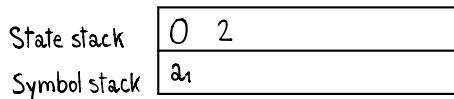
Possiamo concludere che la grammatica è LR(1) in quanto la tabella non ha entry multiply-defined.

Vediamo come si effettua il **riconoscimento** della stringa $a_1 a_2 b_2 b_1$. Quando inizio, nell'automa sono nello stato 0.

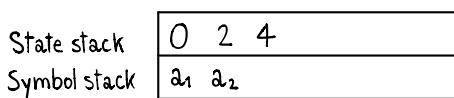


Nell'input buffer leggo a_1 .

La tabella, alla entry [0, a], indica "shift 2". Metto a_1 nel symbol stack e 2 nello state stack. Nell'input buffer leggo a_2 .



La tabella, alla entry [2, a], indica "shift 4". Metto a_2 nel symbol stack e 4 nello state stack. Nell'input buffer leggo b_2 .



La tabella, alla entry [4, b], indica "reduce S-> ε ". Tolgo un quantitativo nullo di elementi dagli stack. Metto S nel symbol stack. Nello state stack metto lo stato indicato dalla entry [4, S], cioè 6.

State stack	0 2 4 6
Symbol stack	a ₁ a ₂ S

La tabella, alla entry [6, b], indica "shift 7". Metto b₂ nel symbol stack e 7 nello state stack. Nell'input buffer leggo b₁.

State stack	0 2 4 6 7
Symbol stack	a ₁ a ₂ S b ₂

La tabella, alla entry [7, b], indica "reduce S->aSb". Tolgo 3 elementi dagli stack. Metto S nel symbol stack. Nello state stack metto lo stato indicato dalla entry [2, S], cioè 3.

State stack	0 2 3
Symbol stack	a ₁ S

La tabella, alla entry [3, b], indica "shift 5". Metto b₁ nel symbol stack e 5 nello state stack. Nell'input buffer leggo \$.

State stack	0 2 3 5
Symbol stack	a ₁ S b ₁

La tabella, alla entry [5, \$], indica "reduce S->aSb". Tolgo 3 elementi dagli stack. Metto S nel symbol stack. Nello state stack metto lo stato indicato dalla entry [0, \$], cioè 1.

State stack	0 1
Symbol stack	S

La tabella, alla entry [1, \$], indica "accept".

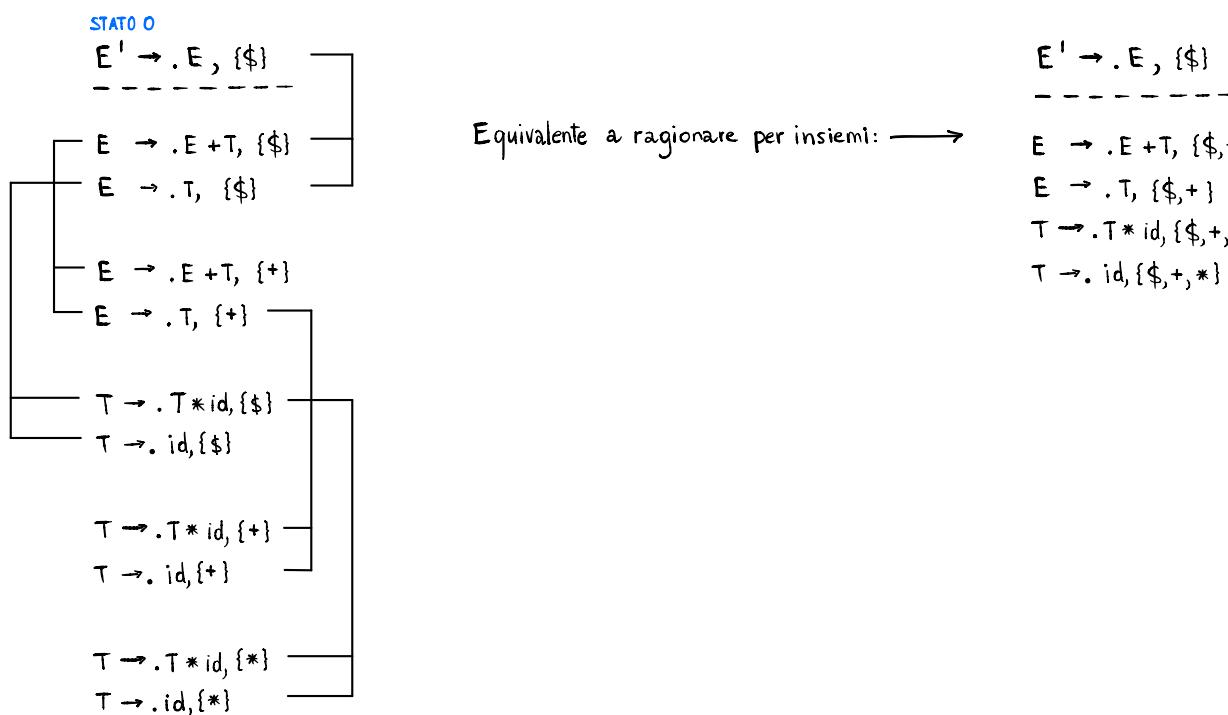
Esercizio 2

Grammatica:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * id \mid id \end{aligned}$$

Vogliamo stabilire se la grammatica sia analizzabile con il parsing LR canonico.

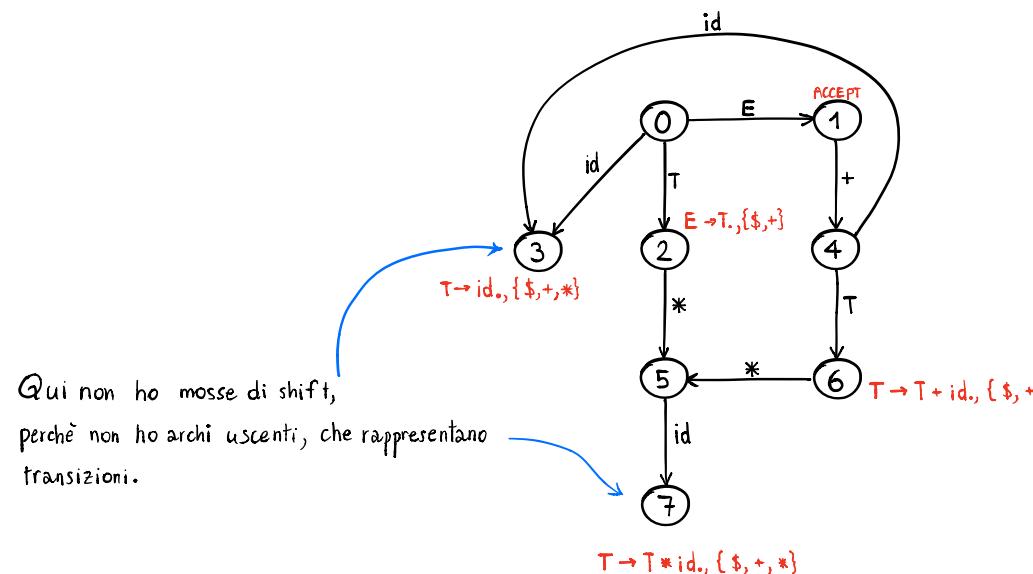
Costruiamo la collezione degli stati che servono all'automa caratteristico.



Grammatica:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * id \mid id \end{aligned}$$

Automa caratteristico:



R/R conflict: la stessa entry della tabella contiene “reduce $A \rightarrow \beta$ ” e “reduce $A \rightarrow \gamma$ ” per $\beta \neq \gamma$. Condizione necessaria per trovare nella tabella di parsing un conflitto R/R è di avere in un certo stato dell'automa caratteristico più reducing item (è l'unica situazione per la quale, nella tabella, andrei a mettere due reduce). Poi, bisognerebbe anche avere che il lookahead set associato a ciascun reducing item abbiano intersezione non nulla.

S/R conflict: la stessa entry della tabella contiene “shift P” (P è uno stato) e “reduce $A \rightarrow B$ ”, per qualche P e qualche A $\rightarrow B$.

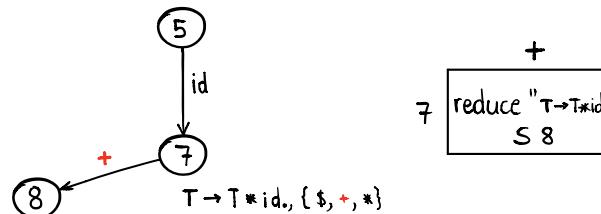
Condizione necessaria per trovare nella tabella di parsing un conflitto S/R è di avere in un certo stato dell'automa caratteristico una transizione uscente etichettata con un certo terminale ed un reducing item il cui lookahead set contiene quel terminale.

Sulla base di queste considerazioni, possiamo concludere che la tabella, che si ricaverebbe dal precedente automa caratteristico e dalle informazioni appuntate in rosso sui reducing item e i rispettivi lookahead set, non avrà né conflitti R/R né conflitti S/R.

Tabella di parsing:

	id	+	*	\$	E	T
0	S3				1	2
1		S4			ACC	
2	r "E->T"		S5		r "E->T"	
3	r "T->id"	r "T->id"		r "T->id"		
4	S3					6
5	S7					
6		r "E->E+T"	S5		r "E->E+T"	
7		r "T->T*id"	r "T->T*id"	r "T->T*id"		

Se l'automa caratteristico fosse stato così, avrei avuto un S/R conflict:



Tutt'altro paio di maniche se prendiamo una grammatica ambigua:

$E \rightarrow E + E \mid E * E \mid id$

È ambigua perché non dice che il $+$ è associativo a sinistra, che il $*$ è associativo a sinistra, che il $*$ ha la precedenza sul $+$. La grammatica di prima lo diceva.

Stato 0:

$E' \rightarrow .E, \{ \$ \}$

- - - - -

$E \rightarrow .E+E, \{ \$, +, * \}$

$E \rightarrow .E*E, \{ \$, +, * \}$

$E \rightarrow .id, \{ \$, +, * \}$

Stato 1:

$E' \rightarrow E., \{ \$ \}$

$E \rightarrow E.+E, \{ \$, +, * \}$

$E \rightarrow E.*E, \{ \$, +, * \}$

Automata caratteristico:

Stato 2:

$E' \rightarrow id., \{ \$, +, * \}$

Stato 3:

$E \rightarrow E+.E, \{ \$, +, * \}$

- - - - - - -

$E \rightarrow .E+E, \{ \$, +, * \}$

$E \rightarrow .E*E, \{ \$, +, * \}$

$E \rightarrow .id, \{ \$, +, * \}$

Stato 4:

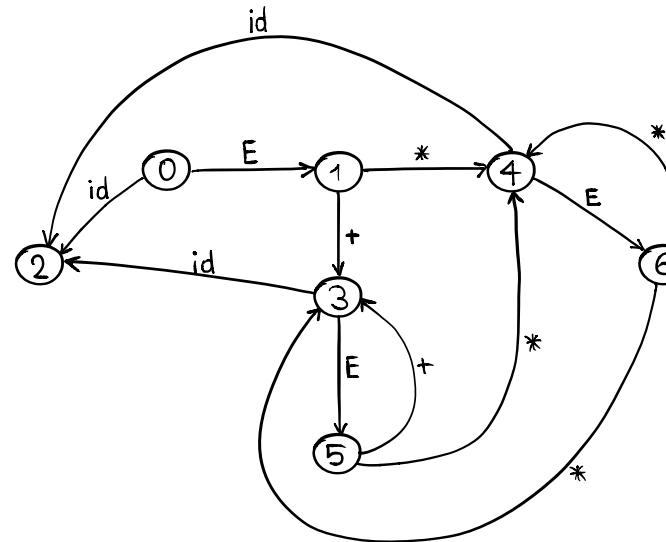
$E \rightarrow E*E.E, \{ \$, +, * \}$

- - - - - - -

$E \rightarrow .E+E, \{ \$, +, * \}$

$E \rightarrow .E*E, \{ \$, +, * \}$

$E \rightarrow .id, \{ \$, +, * \}$



Stato 5:

$E \rightarrow E+E., \{ \$, +, * \}$

$E \rightarrow E.+E, \{ \$, +, * \}$

$E \rightarrow E.*E, \{ \$, +, * \}$

Stato 6:

$E \rightarrow E*E., \{ \$, +, * \}$

$E \rightarrow E.+E, \{ \$, +, * \}$

$E \rightarrow E.*E, \{ \$, +, * \}$

Tabella di parsing:

	id	+	*	\$	E
0	S2				1
1		S3	S4	ACC	
2		E->id	E->id	E->id	
3	S2				5
4	S2				6
5		S3, E->E+E	S4, E->E+E	E->E+E	
6		S3, E->E*E	S4, E->E*E	E->E*E	

Ci sono 4 S/R conflict marcati in grassetto.

Ora sceglio quale, in ciascuno dei quattro casi di conflict S/R, è la mossa appropriata da prendere.

Supponiamo di dover riconoscere la stringa $id_1 + id_2 + id_3$.

Quando inizio, nell'automa sono nello stato 0.

State stack	0
Symbol stack	

Nell'input buffer leggo id_1 .

La tabella, alla entry [0, id_1], indica "shift 2". Metto id_1 nel symbol stack e 2 nello state stack. Nell'input buffer leggo +.

State stack	0 2
Symbol stack	id_1

La tabella, alla entry [2, +], indica "reduce $E \rightarrow id$ ". Tolgo 1 elemento dagli stack. Metto E nel symbol stack.

Nello state stack metto lo stato indicato dalla entry [0, E], cioè 1.

State stack	0 1
Symbol stack	E

La tabella, alla entry [1, +], indica "shift 3". Metto + nel symbol stack e 3 nello state stack. Nell'input buffer leggo id_2 .

State stack	0 1 3
Symbol stack	E +

La tabella, alla entry [3, id_2], indica "shift 2". Metto id_2 nel symbol stack e 2 nello state stack. Nell'input buffer leggo +.

State stack	0 1 3 2
Symbol stack	E + id_2

La tabella, alla entry [2, +], indica "reduce $E \rightarrow id$ ". Tolgo 1 elemento dagli stack. Metto E nel symbol stack.

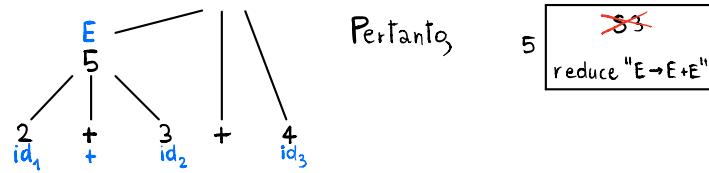
Nello state stack metto lo stato indicato dalla entry [3, E], cioè 5.

State stack	0 1 3 5
Symbol stack	E + E

La tabella, alla entry [5, +], contiene un **S/R conflict**. Come facciamo?

La mossa di reduce sarebbe quella giusta, in quanto noi assumiamo che la grammatica sia associativa a sinistra.

Ese.: $2 + 3 + 4$

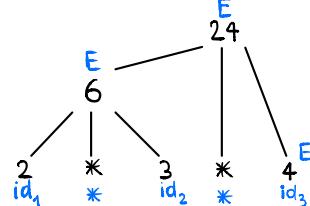


Pertanto,

5	+ SS reduce "E → E + E"
---	--

Supponiamo di dover riconoscere la stringa $id_1 * id_2 * id_3$. La tabella, alla entry [6, *], contiene un **S/R conflict**.

Ese.: $2 * 3 * 4$

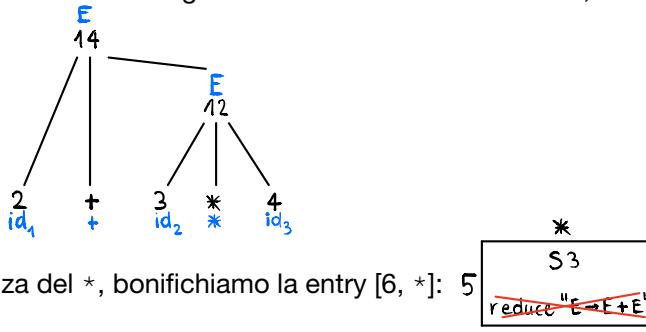


Per esprimere l'associatività a sinistra del *, bonifichiamo in maniera analoga la entry [6, *]:

6	* SS reduce "E → E * E"
---	--

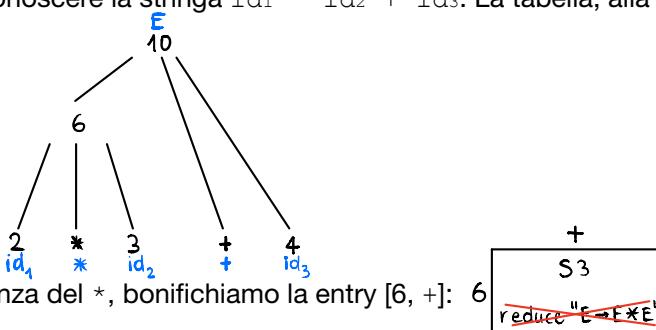
Supponiamo di dover riconoscere la stringa $\text{id}_1 + \text{id}_2 * \text{id}_3$. La tabella, alla entry [5, *], contiene un **S/R conflict**.

Ese.: $2 + 3 * 4$



Per esprimere la precedenza del *, bonifichiamo la entry [6, *]: 5

Ese.: $2 * 3 + 4$



Per esprimere la precedenza del *, bonifichiamo la entry [6, +]: 6

Tabella di parsing “depurata”:

	id	+	*	\$	E
0	S2				1
1		S3	S4	ACC	
2		E->id	E->id	E->id	
3	S2				5
4	S2				6
5		E->E+E	S4	E->E+E	
6		S3	E->E*E	E->E*E	

Qual è il vantaggio clamoroso che le grammatiche ambigue hanno rispetto alle altre?
Sono estremamente sintetiche e human-readable.

Iniziamo con un sommario di quello che abbiamo visto con il parsing canonical LR(1).

Se la tabella associata non contiene conflitti, allora la grammatica è LR. Se invece essa contiene una entry multiply-defined, cioè contenente più direttive, il parsing LR non potrebbe mai essere deterministico, e quindi la grammatica non è LR.

Un po' di terminologia

Item: ha la forma $A \rightarrow \alpha.\beta$

Proj([$A \rightarrow \alpha.\beta, \Delta$]): è la prima componente, cioè $A \rightarrow \alpha.\beta$.

Con $\text{proj}(\mathcal{I}) = \bigcup_{i \in \mathcal{I}} \text{proj}(i)$ intendiamo la proiezione di un certo insieme di item \mathcal{I} .

I kernel item servono a discriminare se, durante la costruzione, uno stato è già stato inserito. Sono kernel item tutti quelli che non compaiono nella chiusura. Sono considerati kernel item quello iniziale: $S' \rightarrow .S$ e tutti quelli dalla forma $A \rightarrow \alpha.\beta$, con $\alpha \neq \beta$.

Con $\text{kernel}(\mathcal{I})$ intendiamo l'insieme dei kernel item di un certo insieme di item \mathcal{I} .

Un po' di osservazioni

Supponiamo di avere due stati \mathcal{I} e \mathcal{J} , cioè due collezioni di item, di un qualche automa caratteristico.

Supponiamo anche che $\text{proj}(\text{kernel}(\mathcal{I})) = \text{proj}(\text{kernel}(\mathcal{J}))$, ossia, l'insieme delle prime componenti dei kernel item di \mathcal{I} è esattamente uguale all'insieme delle prime componenti dei kernel item di \mathcal{J} .

Cosa possiamo dire di \mathcal{I} e di \mathcal{J} ? Sono uguali? **No, dipende dal lookahead set Δ .**

Banalmente, uno prende $S \rightarrow aSb, \Delta_1$ e $S \rightarrow aSb, \Delta_2$. Se Δ_1 e Δ_2 sono diversi, allora \mathcal{I} è diverso da \mathcal{J} .

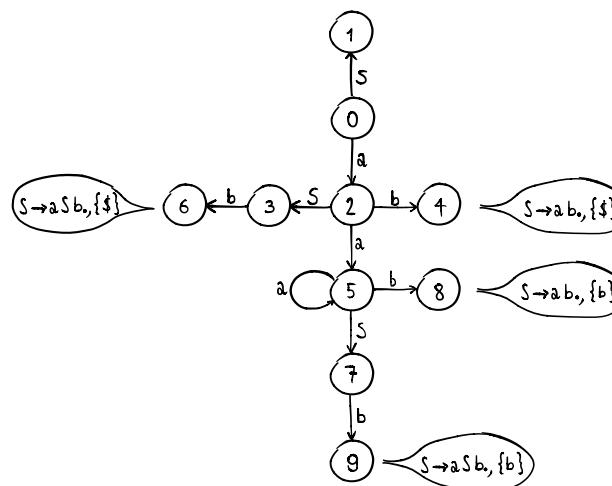
Quindi, cosa possiamo dire di \mathcal{I} e di \mathcal{J} ? È invece vero o no che $\text{proj}(\mathcal{I}) = \text{proj}(\mathcal{J})$?

Vediamo. Supponiamo di avere la grammatica $S \rightarrow aSb \mid ab$. I nostri \mathcal{I} e \mathcal{J} sono:

$S \rightarrow a.Sb, \Delta_1$	$S \rightarrow a.Sb, \Delta_2$
- - - - -	- - - - -
$S \rightarrow .aSb, \{b\}$	$S \rightarrow .aSb, \{b\}$
$S \rightarrow .ab, \{b\}$	$S \rightarrow .ab, \{b\}$

Se $\text{proj}(\text{kernel}(\mathcal{I})) = \text{proj}(\text{kernel}(\mathcal{J}))$, allora $\text{proj}(\text{closure}_1(\text{kernel}(\mathcal{I}))) = \text{proj}(\text{closure}_1(\text{kernel}(\mathcal{J})))$.

Riprendiamo la grammatica $S \rightarrow aSb \mid ab$. L'automa caratteristico era:



Durante la costruzione di questo automa caratteristico, era emerso che gli stati 2 e 5 hanno la caratteristica di avere la medesima proiezione:

Stato 2:

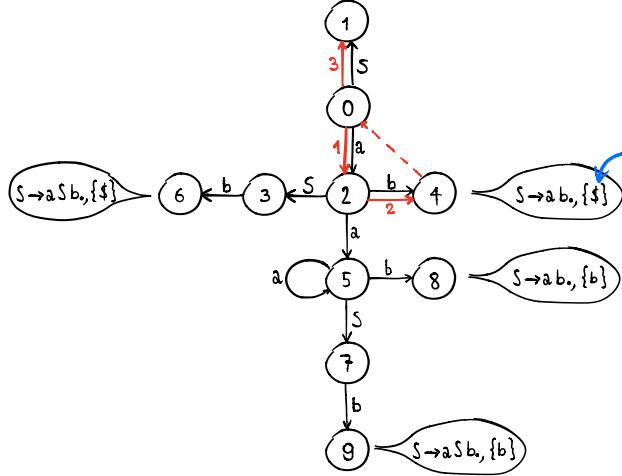
$S \rightarrow a.Sb, \{\$\}$
$S \rightarrow a.b, \{\$\}$
- - - - -

Stato 5:

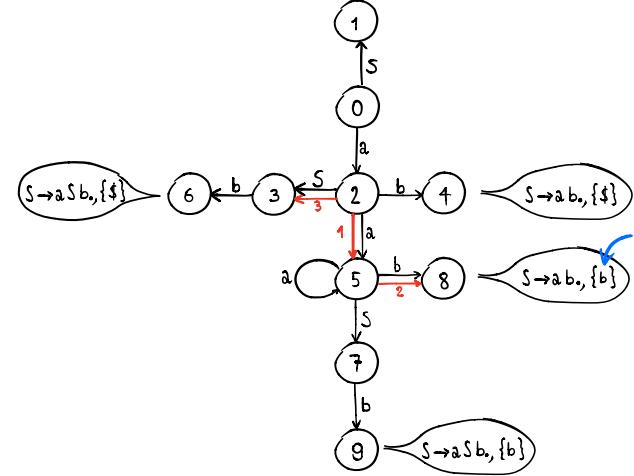
$S \rightarrow a.Sb, \{b\}$
$S \rightarrow a.b, \{b\}$
- - - - -

La maggior parte dei linguaggi di programmazione sono scritti in una qualche grammatica che può essere analizzata nella cosiddetta tecnica LALR. Questa tecnica si può introdurre in modo naturale come un derivato del canonical LR, in cui si vanno ad unificare gli stati che hanno la medesima prima proiezione.

In effetti, se guardiamo l'automa caratteristico di questa grammatica, ci accorgiamo dei seguenti "pattern", in due casi chiaramente diversi:



QUESTO PATTERN CORRISPONDE AL RICONOSCIMENTO
DELLA PAROLA "ab". NOTIAMO CHE DIETRO HO IL '\$'



QUESTO PATTERN CORRISPONDE AL RICONOSCIMENTO DELLA PORZIONE CENTRALE
DI UNA PAROLA COME "aaa**a**bbbb" (IN BLU). NOTIAMO CHE DIETRO HO LA 'b'

Il nostro obiettivo è ridurre il numero di stati dell'automa caratteristico.

Per conseguire ciò, possiamo scrivere grammatiche di classe LALR.

Le grammatiche di classe LALR hanno tabelle di parsing più piccole.

Tali tabelle possono essere generate in maniera non efficiente: prima facendo l'automa caratteristico LR e la relativa tabella di parsing LR, poi facendo l'unione degli stati con le stesse proiezioni.

Nel nostro esempio, dallo stato 2 e dallo stato 5 costruisco uno stato che ha la prima proiezione di 2 e di 5 e come lookahead set l'unione dei lookahead set di 2 e di 5. Sto dicendo che il 2 e il 5 diventano un unico stato. Dopodiché, visto che lo stato 2 e lo stato 5 avevano una b-transizione rispettivamente allo stato 4 e allo stato 8, cosa succede all'automa caratteristico? Ragioniamo: se 2 e 5 hanno la stessa proiezione, allora 4 e 8 hanno la stessa proiezione, in quanto 4 e 8 erano stati generati da due stati con la stessa proiezione del kernel.

Un ragionamento analogo ci conduce a mettere insieme le seguenti coppie di stati: 2 e 5, 4 e 8, 3 e 7, 6 e 9.

Guardiamo la tabella di parsing:

	a	b	\$	S
0	S 2			1
1			acc	
2	S 5	S 4		3
3		S 6		
4			r S->ab	
5	S 5	S 8		7
6			r S->aSb	
7		S 9		
8		r S-> ab		
9		r S-> aSb		

Abbiamo detto, lo stato 2 e lo stato 5 hanno la stessa proiezione del kernel.

E infatti, se guardiamo la tabella, 2 e 5 hanno in comune shift-moves per gli stessi elementi.

Visualizziamo le prime proiezioni in comune:

6: S -> aSb., { \$ }

9: S -> aSb., { b }

3: S -> aS.b, { \$ }

7: S -> aS.b, { b }

Notiamo che stati con la medesima prima proiezione
hanno il '.' davanti al medesimo simbolo.

4: S -> ab., { \$ }

8: S -> ab., { b }

Procediamo a fare il merge degli stati simili.

Nella tabella, facciamo il join delle informazioni che abbiamo.

	a	b	\$	S
0	s 2			1
1			acc	
25	s 25	s 48		37
37		s 69		
48	r S->ab	r S->ab		
69	r S->aSb	r S->aSb		

Sostanzialmente,

I: $A \rightarrow \alpha \cdot, \Delta_1$

J: $A \rightarrow \alpha \cdot, \Delta_2$

IJ: $A \rightarrow \alpha \cdot, \Delta_1 \cup \Delta_2$

Abbiamo notato che la tabella si è ridotta di molto.

Ribadiamo che i linguaggi di programmazione si scrivono in modo tale che rientrino nella classe di grammatiche analizzabili con la tecnica LALR. Si dice che *una grammatica è LALR* se la tabella di ottenuta con la modalità LALR non contiene conflitti.

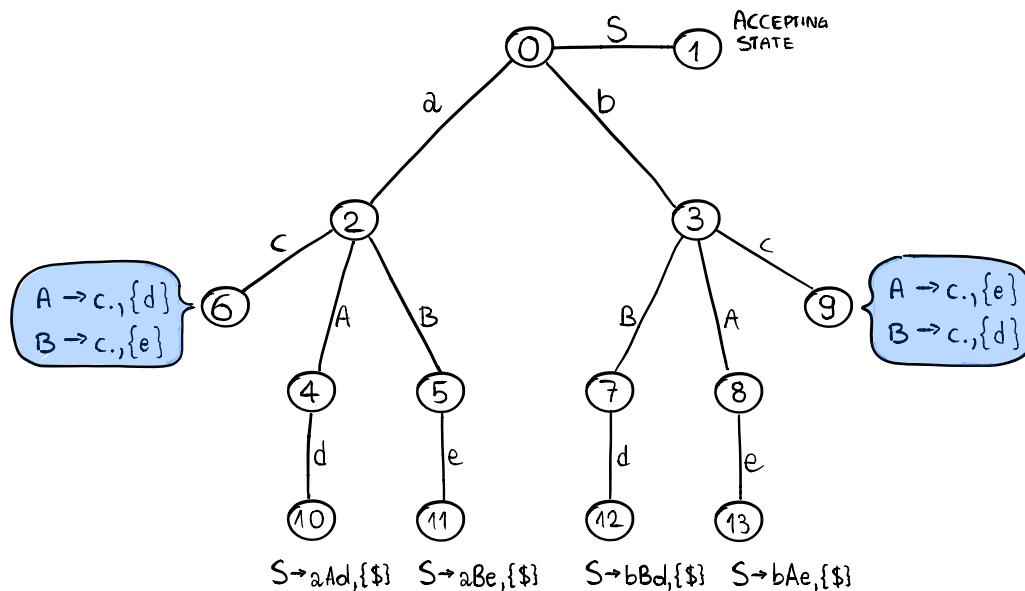
Tuttavia, noi abbiamo fatto l'esercizio nella maniera più inefficiente possibile, cioè prima costruendo la tabella di parsing. Con calma, arriveremo a vedere algoritmi un po' più complessi che arrivano direttamente a generare la tabella LALR.

Ora prendiamo la seguente grammatica, che abbiamo visto parzialmente qualche giorno fa.

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$



C'è qualcosa che unifichiamo? Sì, lo stato 6 e lo stato 9, che hanno le stesse prime proiezioni. E come è fatta la tabella alla riga 6 e 9?

	e	d
6	r B->c	r A->c
9	r A->c	r B->c

Cosa ottengo quando metto insieme? Due R/R conflict! Nella tabella otterrei:

	e	d
69	r B->c	r A->c
	r A->c	r B->c

Quindi, questa grammatica è LR, ma non è LALR.

Visualizziamo questo fatto.

Se io leggo aC , io non so se la c che sto vedendo è un derivato della A o della B , a meno che non guardi dietro di me nell'input buffer. Se guardo nell'input buffer, questo dubbio me lo tolgo subito.

Se nell'input buffer leggo una d , allora so che la c è un derivato della A ; se leggo una e , allora la c è un derivato della B .

Se leggo bC , il problema è analogo. A seconda di quello che leggo nell'input buffer (e oppure d), so che la c è un derivato di A oppure di B .

Avere due stati separati nella tabella, lo stato 6 e lo stato 9, fanno questo servizio di capire da dove sto arrivando.

Se invece metto insieme, perdo la nozione della provenienza che avevo prima.

Disquisizione sui lookahead set nel parsing LR

Nel parsing LR, i lookahead set che noi trasportiamo da stato a stato servono, in sostanza, per raffinare gli insiemi dei follow del non terminale per il quale facciamo la riduzione, secondo un principio di località; cioè, appunto, di qual è il percorso che abbiamo seguito per arrivare allo stato nel quale facciamo la riduzione.

Quando facciamo il merge degli stati e otteniamo una tabella LALR, l'azione di unire i lookahead set di item che hanno la stessa proiezione significa che aumentiamo il numero di elementi per cui siamo disposti a fare una riduzione per una certa produzione.

Per qual motivo si vanno a scomodare queste tecniche così sofisticate?

Banalmente, potremmo semplificarcici di molto la vita, mettendo d'ufficio, a ogni passo, i follow dei driver della produzione, anziché portarci avanti i lookahead set. Questa sarebbe la tecnica SLR (Simple LR).

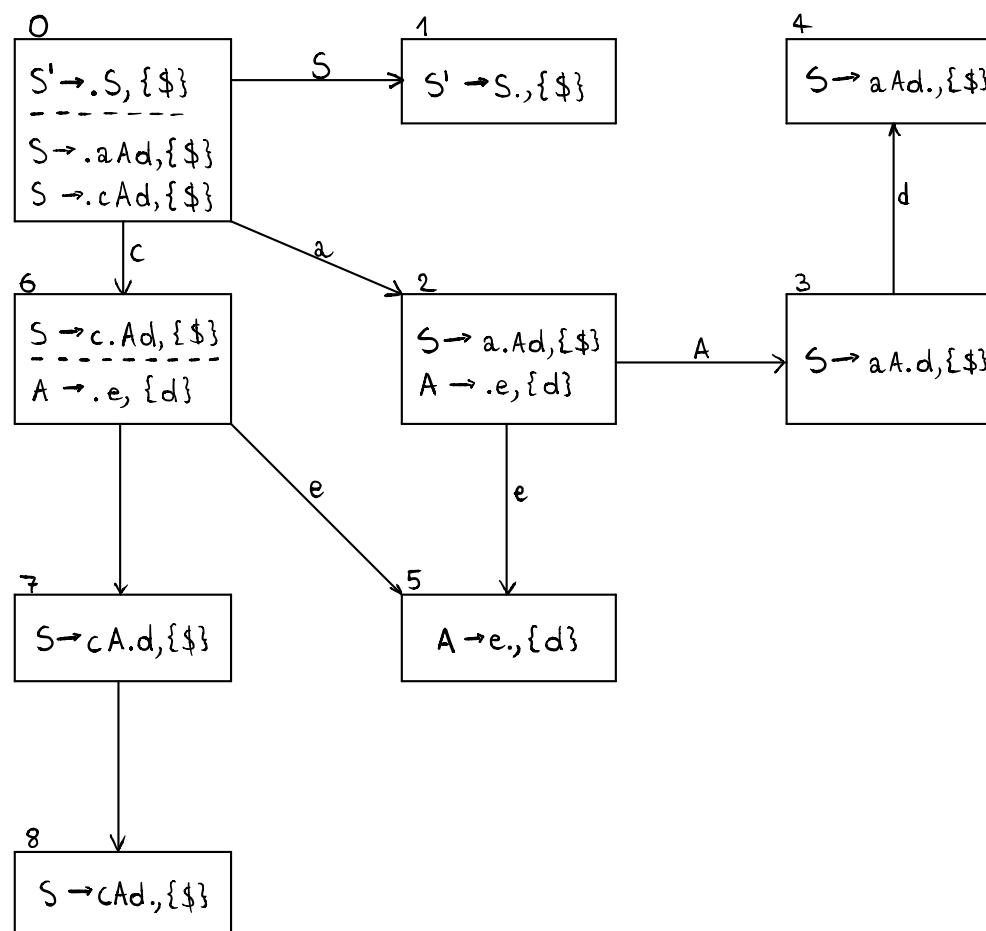
Peccato che, se così facciamo, il nostro linguaggio non può avere i puntatori e una serie di altre feature, che con SLR non c'è verso di esprimere.

E c'è un'altra serie di casi simili a questo.

Grammatica:

$S \rightarrow aAd \mid cAd$

$A \rightarrow e$



Rivediamo quello che abbiamo cominciato ieri, cioè la costruzione dell'automa caratteristico. Usiamo un esempio che è un classico, in quanto include la nozione dei puntatori.

Partiamo dalla grammatica G:

$S \rightarrow L = R \mid R$
 $L \rightarrow * R \mid id$
 $R \rightarrow L$

$L = R$ è l'assegnamento.

L può essere un puntatore ad R : $L \rightarrow R$, oppure un id .

Facciamo l'automa caratteristico per la grammatica, poi vediamo come da questo, facendo il merging degli stati, possiamo ottenere l'automa per il caso LALR.

Stato 0:

$S' \rightarrow .S, \{\$\}$
- - - - -

Chiudo per S rispetto al $\$$:

$S \rightarrow .L=R, \{\$\}$

$S \rightarrow .R, \{\$\}$

Chiudo sia per la L che per la R portandomi dietro il lookahead $\$$

$L \rightarrow .*R, \{=\}$

$L \rightarrow .id, \{=\}$

$R \rightarrow .L, \{\$\}$

La chiusura non è completata, in quanto ho inserito un item $R \rightarrow .L, \{\$\}$. Aggiorno i seguenti item:

$L \rightarrow .*R, \{=,\$\}$

$L \rightarrow .id, \{=,\$\}$

Stato 1: $\tau(0, S)$: si legge "è la transizione dello stato 0 rispetto alla S ":

$S' \rightarrow S., \{\$\}$

Stato 2:

$S \rightarrow L.=R, \{\$\}$

$R \rightarrow L., \{\$\}$

Stato 3: $\tau(0, R)$

$S \rightarrow R., \{\$\}$

Stato 4: $\tau(0, *)$: ...e, vedi dopo, $\tau(4, *)$:

$L \rightarrow *.R, \{=,\$\}$
- - - - -

Dobbiamo chiudere la R rispetto al lookahead set $\{=,\$\}$:

$R \rightarrow .L, \{=,\$\}$

Dobbiamo chiudere la L rispetto al lookahead set $\{=,\$\}$:

$L \rightarrow .*R, \{=,\$\}$

$L \rightarrow .id, \{=,\$\}$

Stato 5: $\tau(0, id)$: ...e, vedi dopo, $\tau(4, id)$:

$L \rightarrow id., \{=,\$\}$

Dallo stato 2 abbiamo una transizione rispetto ad $=$.

Stato 6: $\tau(2, =)$:

$S \rightarrow L=.R, \{\$\}$
- - - - -

Dobbiamo chiudere la R :

$R \rightarrow .L, \{\$\}$

Dobbiamo chiudere la L :

$L \rightarrow .*R, \{\$\}$

$L \rightarrow .id, \{\$\}$

Dallo stato 4 abbiamo una transizione rispetto alla R , una rispetto alla L , una rispetto ad $*$, una rispetto ad id :

Stato 7: $\tau(4, R)$:

$L \rightarrow *R., \{=,\$\}$

Stato 8: $\tau(4, L)$:

$R \rightarrow L., \{=, \$\}$

Quando consideriamo la transizione dallo stato 4 rispetto ad $*$, abbiamo un kernel item $L \rightarrow * . R, \{=\$,\}$, che è esattamente quello dello stato 4. Quindi, il target della $*$ -transizione dallo stato 4 è lo stato 4 medesimo. Aggiungiamo allo stato 4: $\tau(4, *)$.

Quando consideriamo la transizione dallo stato 4 rispetto ad id , abbiamo un kernel item $L \rightarrow id., \{=, \$\}$, che è esattamente quello dello stato 5. Aggiungiamo allo stato 5: $\tau(4, id)$.

Dallo stato 6 abbiamo una transizione rispetto alla R , una rispetto alla L , una rispetto ad $*$, una rispetto ad id :

Stato 9: $\tau(6, R)$:

$S \rightarrow L=R., \{\$\}$

Stato 10: $\tau(6, L)$: ...e, vedi dopo, $\tau(11, L)$:

$R \rightarrow L., \{\$\}$ //Noto che la proiezione è uguale a quella dello stato 8, ma il lookahead set è diverso

Stato 11: $\tau(6, *)$: ...e, vedi dopo, $\tau(11, *)$ e $\tau(11, id)$.

$L \rightarrow * . R, \{\$\}$ //Noto che la proiezione è uguale a quella dello stato 7, ma il lookahead set è diverso

Dobbiamo chiudere la R :

$R \rightarrow .L, \{\$\}$

Dobbiamo chiudere la L :

$L \rightarrow .*R, \{\$\}$

$L \rightarrow .id, \{\$\}$

Stato 12: $\tau(6, id)$:

$L \rightarrow id., \{\$\}$ //Noto che la proiezione è uguale a quella dello stato 5, ma il lookahead set è diverso

Dallo stato 11 abbiamo una transizione rispetto alla R , una rispetto alla L , una rispetto a $*$

Stato 13: $\tau(11, R)$:

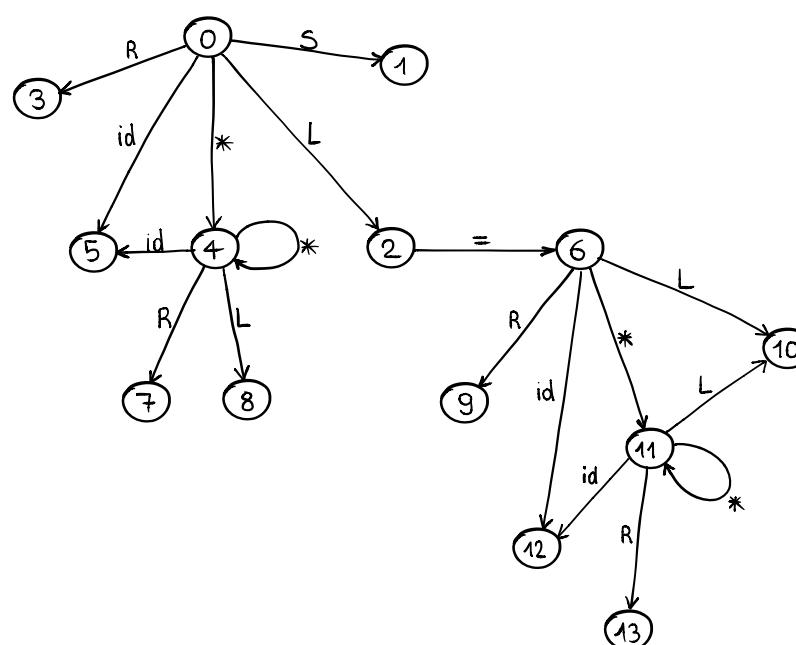
$L \rightarrow *R., \{\$\}$ //Noto che la proiezione è uguale a quella dello stato 7, ma il lookahead set è diverso

Quando consideriamo la transizione dello stato 11 rispetto alla L , abbiamo un kernel item $R \rightarrow L., \{\$\}$, che è esattamente quello dello stato 10. Aggiungiamo allo stato 10: $\tau(11, L)$.

Quando consideriamo la transizione dello stato 11 rispetto a $*$, abbiamo un kernel item $L \rightarrow .*R, \{\$\}$, che è esattamente quello dello stato 11 stesso. Aggiungiamo allo stato 11: $\tau(11, *)$.

Quando consideriamo la transizione dello stato 11 rispetto a id , abbiamo un kernel item $L \rightarrow id., \{\$\}$, che è esattamente quello dello stato 12. Aggiungiamo allo stato 12: $\tau(11, id)$.

Automa:



Per costruire l'automa unificato (o *merged automa*), e quindi arrivare a costruire la tabella LALR di questa grammatica, dobbiamo innanzitutto individuare gli *stati equivalenti*.

Il linguaggio generato dalla grammatica comprende:

- Stringhe del tipo $***\dots*id$,
- Stringhe contenenti uno e un solo $=$.

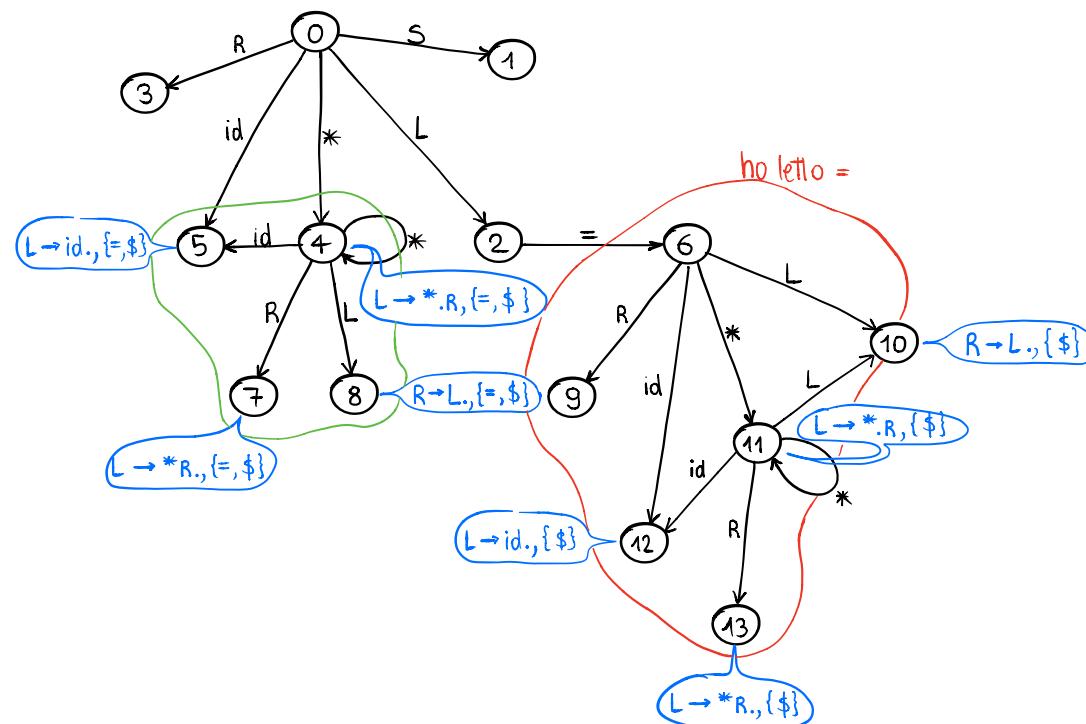
Poniamo l'attenzione sugli stati 5 ed 8.

Lo stato 5, con l'item $L \rightarrow id., \{=, \$\}$, dice: "Sono disposto a dire che questo *id* viene dalla *L* se dopo di me vedo o l' $=$ o il $\$$ ". Questo vuol dire che, se sono nello stato 5, posso o trovarmi alla fine della stringa, o essere l'*id* immediatamente a sinistra dell' $=$.

Lo stato 12, con l'item $L \rightarrow id., \{\$\}$, dice: "Sono disposto a dire che questo *id* viene dalla *L* solo se dopo di me vedo il $\$$ ". Se sono nello stato 12 vuol dire che ho già avuto occasione di scoprire lo stato 5.

Abbiamo che gli stati 5 ed 8 parlano della stessa cosa, ma in momenti distinti del riconoscimento di una stringa.

Questo ragionamento vale per tutte le seguenti coppie di stati: 10-8, 11-4, 12-5, 13-7.



Questi stati sono quelli che metteremo insieme per fare il merged automa.

Dunque, l'idea è quella di ottenere partizioni di stati equivalenti in cui la nozione di equivalenza riguarda la prima proiezione.

Merged automata

Sia A l'automa caratteristico per il parsing LR(1).

Sia A_m l'automa ottenuto dal merging degli stati poter ragionare sulle tabelle di parsing LALR(1).

Dobbiamo rispondere a tre diverse domande.

Chi sono gli stati di A_m ?

Stati di A_m : ogni stato di A_m rappresenta la classe di stati di A con medesima prima proiezione.

Gli stati 0, 1, 2, 3, 6, 9 non hanno proiezioni equivalenti.

Due stati come il 10: $R \rightarrow L., \{\$\}$ e l'8: $R \rightarrow L., \{=, \$\}$ hanno medesima proiezione, e confluiscono in un unico stato con $R \rightarrow L., \{\$\} \cup \{=, \$\}$ ossia $R \rightarrow L., \{=, \$\}$.

Quali di sono le transizioni di A_m ?

Siccome stiamo facendo un automa, dobbiamo descrivere come sono le transizioni, che entrano ed escono nei nuovi stati che abbiamo costruito. Come facciamo a rispondere?

Possiamo innanzitutto affermare che le transizioni entranti ed uscenti in e da due stati che hanno la stessa proiezione non dipendono certamente dal lookahead set. (Il lookahead set fa una discriminazione dei possibili FOLLOW dei non terminali che dipendono dal cammino dal quale proveniamo).

Le transizioni uscenti si calcolano andando a guardare chi c'è alla destra del '.' all'interno della proiezione dell'item.

Se uno stato ha una certa transizione, allora per forza la ha anche uno stato equivalente.

Chi è il target? Da due stati che hanno la medesima proiezione (ma lookahead differente), posso: o finire in un unico stato (non c'erano altri stati con la stessa proiezione), o finire in due stati con medesima proiezione (ma lookahead differente), cioè due stati di una stessa classe.

Ribadiamo: le transizioni dell'automa giocano sulla proiezione, il lookahead set non c'entra nulla!

Nell'automa dell'esempio:

Né lo stato 10 né lo stato 8 hanno transizioni e quindi target.

Lo stato 11 e lo stato 4 sono ricchi di transizioni:

- L'11 con una id-transizione va in 12. Il 4 con una id-transizione va in 5. Verifichiamo che c'è (necessariamente) una coppia 12-5. Quindi, lo stato 114 avrà una id-transizione allo stato 125.
- L'11 con una R-transizione va in 13, il 4 con una R-transizione va in 7. Verifichiamo che c'è una coppia 13-7. Quindi, lo stato 114 avrà una R-transizione allo stato 137.
- L'11 con una L-transizione va in 10, il 4 con una L-transizione va in 8. Verifichiamo che c'è una coppia 10-8. Quindi, lo stato 114 avrà una L-transizione allo stato 108.
- L'11 con una *-transizione va in 11, il 4 con una *-transizione va in 4. Verifichiamo che c'è una coppia 11-4. Quindi, lo stato 114 avrà una *-transizione allo stato 114.

Il merging è un'operazione di unione che facciamo per risparmiare sugli stati che sono più o meno simili, ma che hanno lookahead set diversi.

Definiamo formalmente le transizioni di A_m : se lo stato M di A_m è tale che la $\text{proj}(M)$ è uguale a $\text{proj}(L)$ con L stato di A (l'automa originario), e se L ha una Y -transizione ad L' , allora M ha una Y -transizione allo stato M' tale che $\text{proj}(M')$ è uguale a $\text{proj}(L')$.

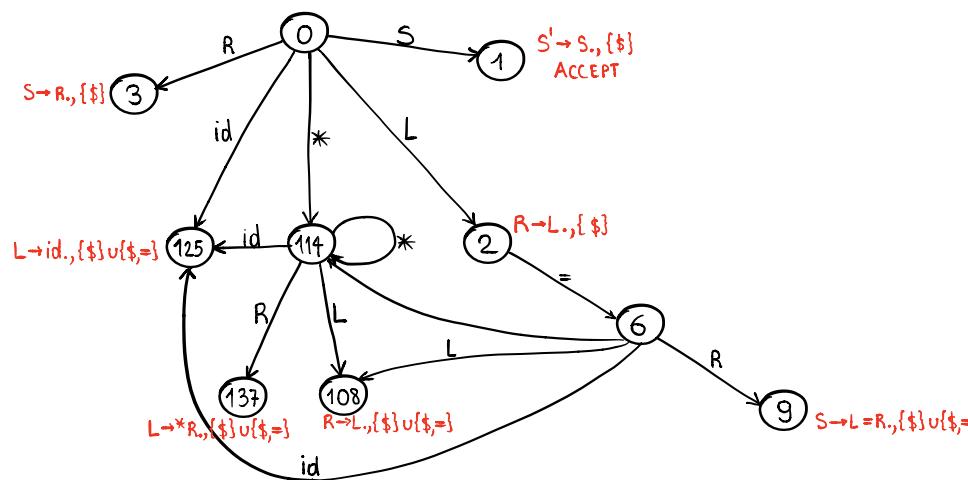
Per capire questa definizione, prova con $M = 114$, $L = 4$, $Y = R$, $L' = 7$.

Come trattiamo i reducing item?

Una volta capito l'automa, siamo in grado di collocare nella tabella le mosse di shift e le mosse di goto.

Ci resta da determinare come collocare le mosse di reduce, che dipendono dai reducing items e dai loro lookahead set.

Disegniamo l'automa che si ottiene, evidenziamo i reducing items.

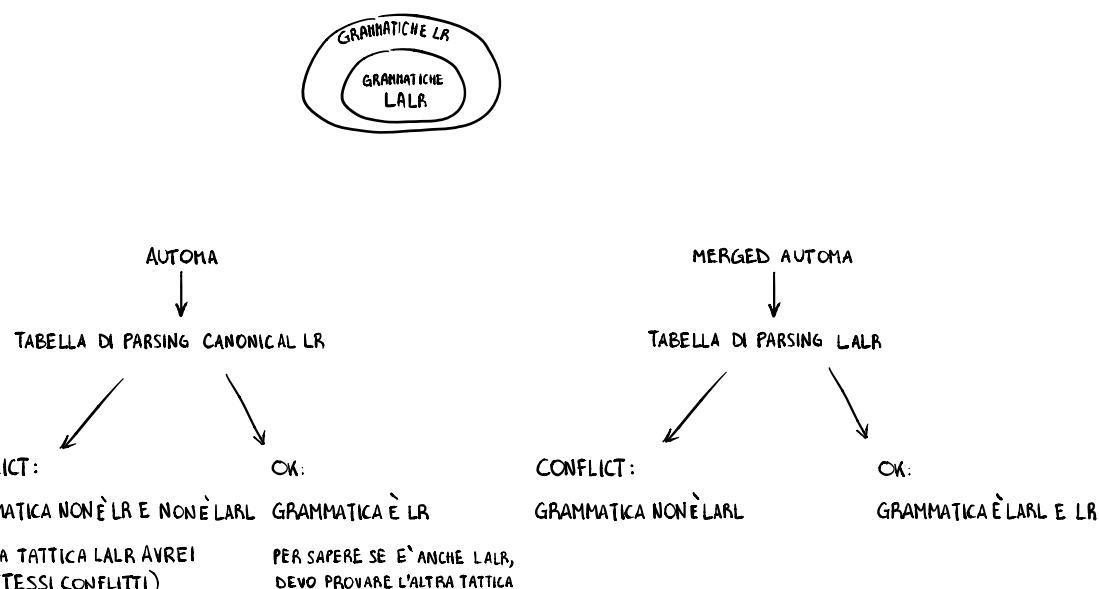


Da questo automa, con quelle informazioni sui reducing item e sui lookahead set, si costruisce, con lo stesso criterio che si adopera per costruire la tabella di parsing LR(1), la tabella di parsing LALR(1).

Quali conflitti possono essere introdotti dall'operazione di merge degli stati?

È possibile che l'operazione di merge degli stati causi l'introduzione di conflitti reduce-reduce, ma mai di shift-reduce.

Ribadiamo: se facciamo questa stessa operazione direttamente sulla tabella, sostituendo le righe, non possiamo mai andare a inserire dei conflitti di shift-reduce, perché l'operazione che qui facciamo, il merge degli stati, è mettere insieme tutti e quanti gli item che appartengono a stati con la stessa proiezione. La mossa dello shift dipende dalle transizioni dell'automa, quindi dipende dalla produzione dell'item. Se noi abbiamo unito due stati, l'unico possibile tipo di conflitto che possiamo generare, che non era possibile nella tabella LR e invece c'è nella tabella LALR, è un conflitto reduce-reduce.



All'inizio del corso eravamo in grado di determinare se la grammatica era context free, context dependent, regolare. Ciò che rende fattibile questa prima grande suddivisione è il fatto che sono proprietà visibili a colpo d'occhio.

L'analizzabilità di una grammatica, invece, non è normalmente visibile. È visibile se la grammatica è a misura d'uomo, ossia se consiste di poche produzioni. Nessuno riesce a dire se una grammatica con numerose produzioni è, ad esempio, LALR.

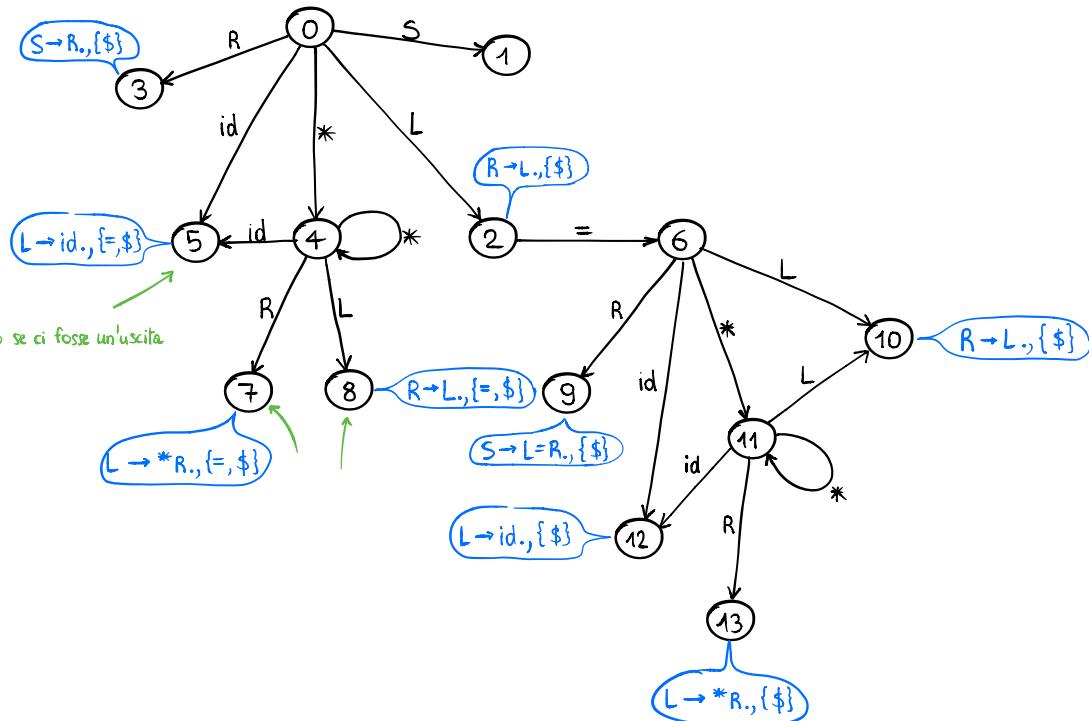
Si parla di classi di grammatiche rispetto alla tecnica che funziona per la loro analisi.

Per dimostrare che una grammatica sta in una certa classe, bisogna verificare che la tabella di parsing per quella classe non contenga entry multiply-defined.

Quindi, si dice che una grammatica è LALR se la tabella di parsing LALR non ha entry multiply-defined.

La grammatica che abbiamo visto oggi è LR?

Possiamo rispondere anche dando un'occhiata all'automa:



Non ho stati che hanno più di un reducing item, con proiezioni diverse e lookahead set uguali \rightarrow No conflitti reduce/reduce.

Non ho stati che hanno sia un reducing item il cui lookahead set contiene un certo simbolo s
che una transizione uscente per s \rightarrow No conflitti shift/reduce.

La risposta è sì: l'automa è LR.

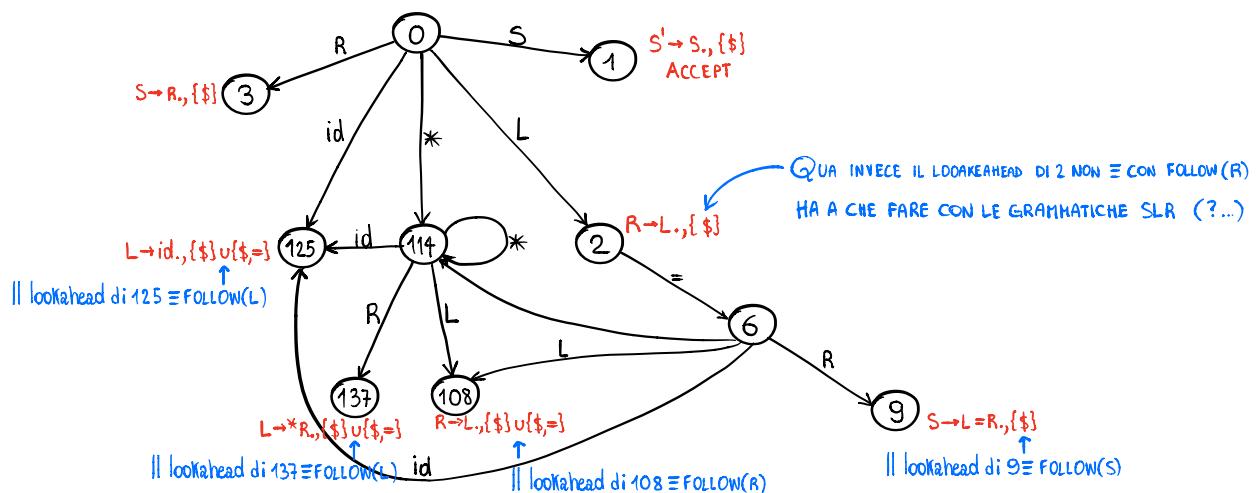
Abbiamo detto che, portandoci dietro negli item le informazioni sui lookahead set, quello che stiamo facendo è andare a guardare proprio quale produzione specifica è stata utilizzata, e quindi dare un flavour di località alla computazione dei FOLLOW.

Detto questo, sia nel caso della computazione dell'automa caratteristico per il parsing canonical LR, che nel caso dell'automa per il parsing LALR, quei lookahead set sono dei sottoinsiemi dei FOLLOW del driver della produzione per la quale riduciamo.

Nella grammatica di oggi, abbiamo i seguenti FOLLOW:

S: \$
L: \$, =
R: \$, =

Diamo un'occhiata all'automa. C'è una stranezza:



Nello stato 2, che rimane tale sia nell'automa LR(1) che nell'automa LALR, noi trovavamo l'item $R \rightarrow L . , \{ \$ \}$.

Sappiamo che un altro possibile FOLLOW di R è $=$.

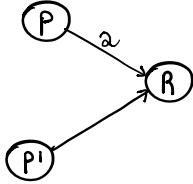
Osservo che possiamo raggiungere lo stato 2 esattamente quando dallo stato 0 abbiamo visto la L , non ancora $L' =$. E infatti, quali sono le stringhe per le quali lì faremo una riduzione? Quelle per le quali abbiamo scelto la produzione $S \rightarrow R$.

Anteprima della lezione successiva: costruzione di un automa caratteristico simbolico che fa l'automa merged on the fly.

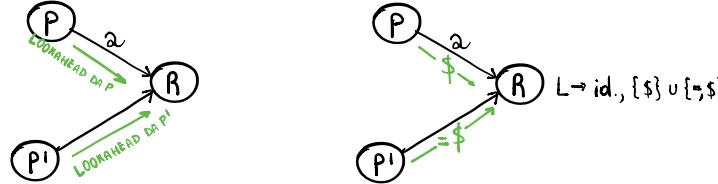
Il procedimento che abbiamo finora seguito era il seguente.

Parto con lo stato iniziale, per poi cominciare a considerare tutte le transizioni da quello stato. Dallo stato iniziale genero altri stati. Per ognuno dei nuovi stati, considero chi sono i target secondo le possibili transizioni. Ma non mi viene detto in che ordine farlo. Infatti, durante il procedimento, capita spesso che venga riconosciuto che uno stato precedentemente generato è il target della transizione da un altro.

Supponiamo di essere partiti da uno stato P . Abbiamo considerato la sua transizione rispetto alla a , e per questo aggiungiamo alla collezione di stati lo stato R . Procediamo e aggiungiamo molti altri stati... Ad un certo punto, ci accorgiamo che R è anche target di un certo stato P' .



Per come funziona l'operazione di merging, l'item in R prende i lookahead set che arrivano da P e da P' :



Il nuovo sistema che vedremo consiste nel fare al volo la costruzione di un automa, che non è istanziato, ma che lascia informazioni simboliche, che riguardano i lookahead set.

Si inizia esattamente come prima, con le solite elucubrazioni sugli item, ma si lasciano in maniera simbolica le informazioni sui possibili lookahead set. Quando si è finita la costruzione, cioè si è certi di aver raccolto tutti i possibili contributi da tutte le possibili parti, perché tutte le transizioni sono state guardate, si prendono le informazioni simboliche che riguardano i lookahead set e si fanno processare ad sistema di equazioni...

Per farci un'idea dell'algoritmo annunciato alla fine della lezione precedente, possiamo subito vederlo in azione.

Questo algoritmo prevede che il lookahead set dei kernel item di uno stato contengano variabili, che, durante l'operazione di chiusura dello stato, sono da considerare come simboli non terminali.

Nell'esempio, sviluppiamo in contemporanea l'automa caratteristico e il sistema di equazioni sulle variabili.

Prendiamo una semplice grammatica LALR, e quindi LR.

$S \rightarrow aSb \mid ab$

Stato iniziale 0:

$S' \rightarrow .S, \{x_0\}$ Nel sistema di equazioni, aggiungo $x_0 = \{\$\}$

$S \rightarrow .aSb, \{x_0\}$

$S \rightarrow .ab, \{x_0\}$

Dallo stato 0 avrò due transizioni, una rispetto alla S e una rispetto alla a .

Rispetto alla S , andiamo nello stato 1:

$S' \rightarrow S., \{x_1\}$ Variabile nuova. La chiusura LR(1) avrebbe prescritto di mettere $\{x_0\}$ come lookahead set $x_1 = x_0$

Rispetto alla a , andiamo nello stato 2:

$S \rightarrow a.Sb, \{x_2\}$ x_2 eredita, cioè che viene da x_0 : $x_2 = x_0$ Nell'operazione di chiusura, x_2 e x_3 sono trattati come terminali

$S \rightarrow a.b, \{x_3\}$ x_3 eredita, cioè che viene da x_0 : $x_3 = x_0$

$S \rightarrow .aSb, \{b\}$ $b = \text{FIRST}(bx_2)$

$S \rightarrow .ab, \{b\}$ $b = \text{FIRST}(bx_2)$

Per lo stato 0 abbiamo considerato tutte le possibili transizioni. Lo stato 1 non ha transizioni.

Dallo stato 2 abbiamo tre transizioni, una per la S , una per la b e una per la a .

Rispetto alla S , andiamo nello stato 3:

$S \rightarrow aS.b, \{x_4\}$ $x_4 = x_2$

Rispetto alla b , andiamo nello stato 4:

$S \rightarrow ab., \{x_5\}$ $x_5 = x_3$

Rispetto alla a , la proiezione del kernel sarebbe:

$S \rightarrow a.Sb, \{b\}$ Dovrei portare dietro l'informazione che il lookahead set è $\{b\}$ $x_2 = x_0 \cup \{b\}$

$S \rightarrow a.b, \{b\}$ Anche da questa parte $x_3 = x_0 \cup \{b\}$

Abbiamo già uno stato che ha questa proiezione del kernel, lo stato 2.

A questo punto, non vogliamo rifare tutta quanta la chiusura: semplicemente, lo riconosciamo, e diciamo che la transizione dallo stato 2 rispetto alla a , torna nello stato 2, ma provoca l'aggiunta di elementi all'insieme rappresentato da x_2 e dall'insieme rappresentato da x_3 .

Per lo stato 2 abbiamo considerato tutte le possibili transizioni.

Dallo stato 3 abbiamo una transizione, per la b .

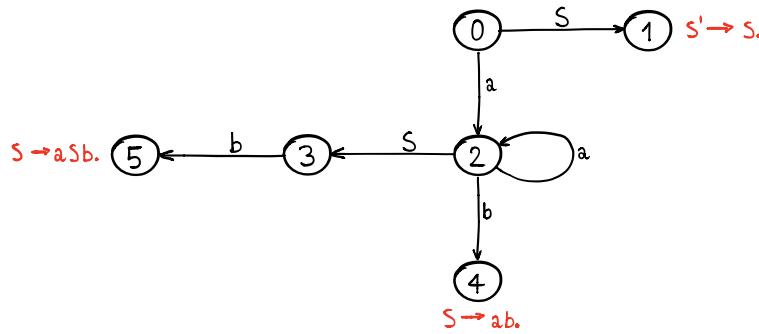
Rispetto alla b , andiamo nello stato 5:

$S \rightarrow aSb., \{x_6\}$ $x_6 = x_4$

Per lo stato 3 abbiamo considerato tutte le possibili transizioni. Lo stato 4 non ha transizioni, e anche lo stato 5.

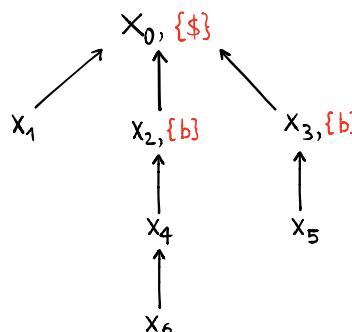
Abbiamo finito.

Disegniamo il layout dell'automa.



Ora dobbiamo capire quale è il valore da mettere nei lookahead set.

Si tratta di realizzare un grafo delle dipendenze. "Dipende" := "La sua definizione usa".
Nei nodi metto anche gli elementi *ground*, cioè quelli non variabili.

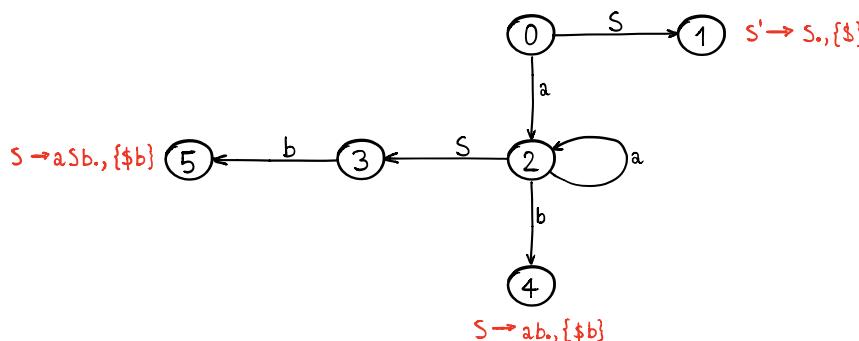


Abbiamo:

$$x_0 = \{ \$ \} = x_1$$

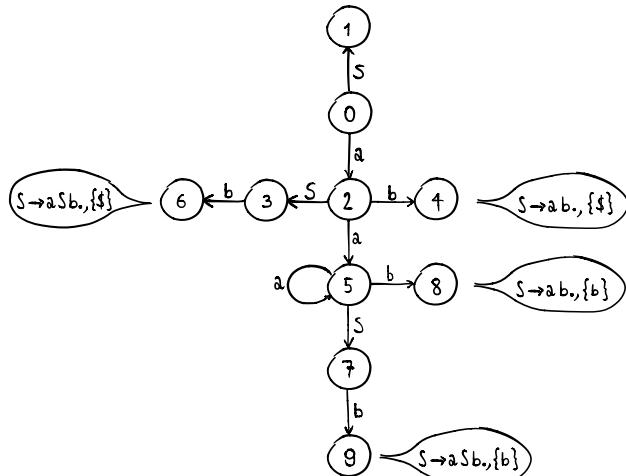
$$x_2 = x_4 = x_6 = x_3 = x_5 = \{ \$, b \}$$

Ora possiamo riempire i lookahead set dell'automa:



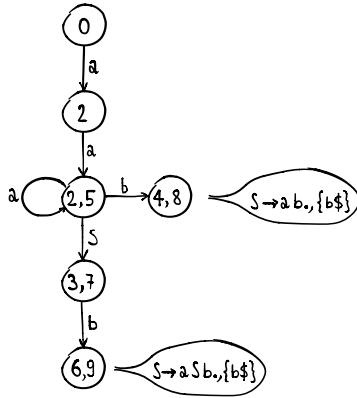
Ora potrei riempire la tabella di parsing: le transizioni per i terminali danno le mosse di shift,
le transizioni per i non-terminali danno le mosse di goto, per i passi di riduzione uso le informazioni dei reducing item
e i loro lookahead set.

Per la medesima grammatica, durante lo studio del parsing LR(1), avevamo ottenuto il seguente automa:



Li stati con la medesima proiezione, cioè quelli che andremmo ad unire se facessimo la costruzione prevista dalla tattica LALR, sono: 3 e 7, 4 e 8, 2 e 5, 6 e 9.

Il layout dell'automa merged è:



Illustriamo brevemente l'algoritmo.

Costruzione dell'automa caratteristico simbolico per parsing LALR(1)

Inizializzare la collezione di stati con $\text{closure}_1(\{[S \rightarrow \cdot S, \{x_0\}]\})$ con x_0 nuova variabile

Inizializzare il sistema di equazioni con $x_0 = \$$

Tag P_0 come non marcato

while c'è uno stato P non marcato nella collezione **do**

 tag P come marcato

foreach Y che compare a destra del marker in un qualche item di P **do** % Fin qua, schema identico a quello del parsing LR(1)

 % Quello che cambia è la gestione delle variabili

 % Computazione temporanea del kernel del Y-target di P

$\text{tmp} = \emptyset$

foreach $[A \rightarrow \alpha.Y\beta, \Delta] \in P$ **do aggiungere** $[A \rightarrow \alpha Y. \beta, \Delta]$ a tmp

 % Distinguiamo i due casi

if $\text{proj}(\text{tmp}) = \text{proj}(\text{kernel}(Q))$ per qualche Q già nella collezione **then** % Abbiamo già generato uno stato con questa proiezione

foreach $[A \rightarrow \alpha Y. \beta, \Delta] \in P, [A \rightarrow \alpha Y. \beta, \{x\}] \in Q$ **do**

 % Aggiungiamo, a tutte quelle le equazioni che sono associate ai vari item del kernel dello stato

 % che abbiamo trovato come target, le componenti che vengono da Δ .

 aggiornare l'equazione $x = \Delta'$ a $x = \Delta' \cup \Delta$

$\tau(P, Y) = Q$

else % Caso in cui generiamo un nuovo stato

foreach $[A \rightarrow \alpha Y. \beta, \Delta] \in \text{tmp}$ **do**

 rimpiazzare Δ in tmp con una nuova variabile x

 inserire $x = \Delta$ nel sistema di equazioni

 generare come non marcato il nuovo stato $Q = \text{closure}_1(\text{tmp})$

$\tau(P, Y) = Q$

Anticipazione della prossima lezione

La prossima lezione vedremo le **syntax directed definitions**, cioè daremo un senso al nostro studio fatto fino ad ora.

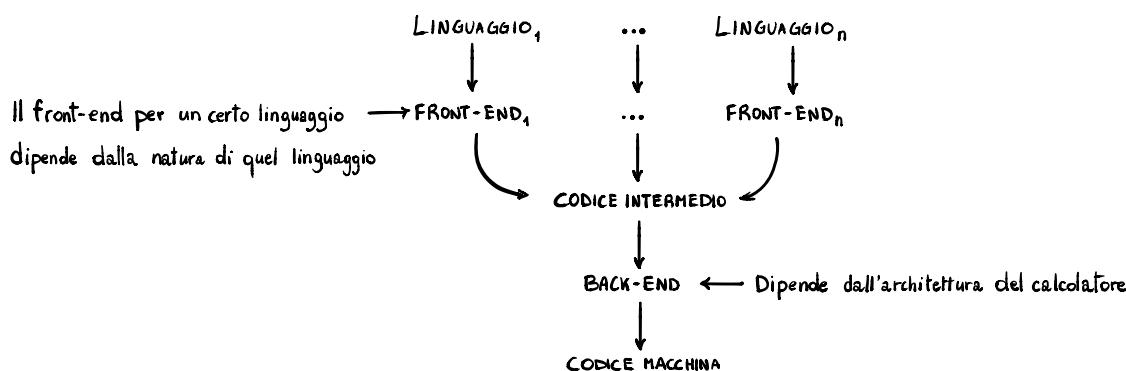
Vedremo la compilazione nella sua forma in assoluto più semplice, che consiste nell'arricchire le grammatiche con delle azioni semantiche. Se noi sappiamo appropriatamente organizzare la grammatica e scrivere le azioni semantiche associate alle produzioni, possiamo conseguire la generazione di codice intermedio, ottenendo così il front-end della compilazione.

Quale è il ruolo delle azioni semantiche? Nell'ambito dell'analisi sintattica bottom-up, ogni qual volta avviene una riduzione di una certa proiezione, viene contestualmente eseguito il codice associato con l'azione semantica di quella produzione. Se noi sappiamo capire quando è che capita un passo di riduzione, sappiamo anche gestire la generazione del codice intermedio.

La compilazione è composta tipicamente da due fasi principali: il front-end e il back-end.

- Il front-end del compilatore si occupa del passaggio dal codice sorgente ad un opportuno codice intermedio. In questa fase hanno luogo l'analisi lessicale, l'analisi sintattica e l'analisi semantica.
- Il back-end del compilatore si occupa del passaggio dal codice intermedio al codice macchina, o codice eseguibile.

Questo schema consente di riutilizzare il medesimo back-end per front-end diversi:



Mentre i linguaggi di programmazione sono numerosissimi, ci sono solo due alternative per la rappresentazione intermedia, che vanno per la maggiore: il *three-address code* e l'*abstract syntax tree*.

Il *three-address code* è il codice per un'architettura astratta di calcolatore.

Il calcolatore astratto è in grado di eseguire semplici istruzioni, caratterizzate da un codice operativo e al più tre indirizzi per gli operandi.

L'*abstract syntax tree*, invece, è la rappresentazione intermedia più usata e più interessante, nonché quella che studieremo.

Anticipazione: Abstract syntax tree

A dispetto del nome *tree*, è un particolare grafo, che deriva direttamente dall'albero di derivazione.

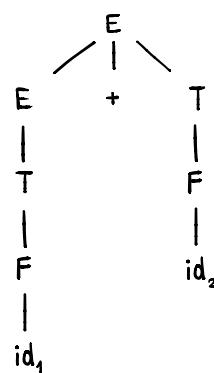
Facciamoci subito un'idea.

Prendiamo la seguente grammatica, che genera il linguaggio delle espressioni aritmetiche:

$E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

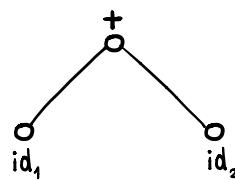
Data una certa espressione aritmetica, quello che ci interessa sapere, alla fine, è il valore della stessa.

Quali sono l'albero sintattico dell'espressione $id_1 + id_2$?



Come detto, quello che ci interessa sapere, alla fine, è la somma fra i due valori id_1 e id_2 .

Quindi, perché mai avere tutti questi nodi in un grafo? Infatti, ad un albero di derivazione come questo, viene fatto corrispondere un albero di derivazione astratto con un certo risparmio di nodi e di archi:



In seguito vedremo che questo grafo (in questo caso, albero) può essere computato mentre facciamo l'analisi sintattica.

Introduciamo ora le syntax-directed definitions

Le syntax-directed definitions si possono agganciare a ogni genere di grammatica, quindi sia a quelle analizzabili in top-down, che quelle analizzabili in bottom-up; motivo per cui è essenziale capire come funzionano le tecniche di parsing, e specialmente in quale momento dell'analisi vengono fatte le scelte di espansione o di riduzione, che ci consentono di derivare l'albero sintattico.

Una **syntax-directed definition** è data da una grammatica context-free **arricchita** da attributi associati ai simboli della grammatica e regole associate alle produzioni della grammatica. Un attribuito può essere: un tipo, un puntatore, una entry in una tabella, una funzione... Più avanti, vedremo come questi attributi possono essere classificati in due categorie principali.

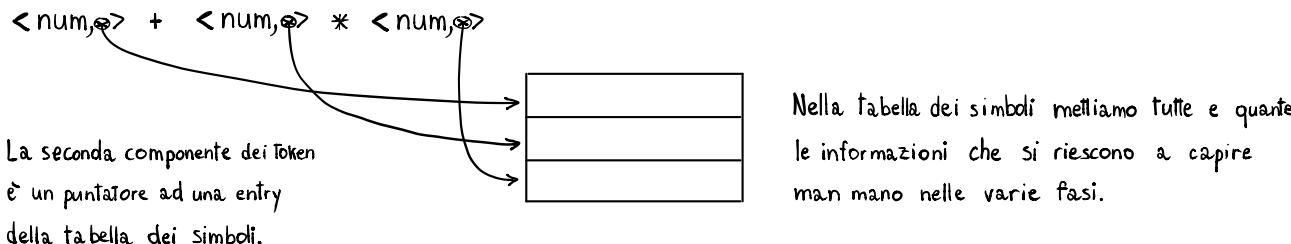
Nel nostro studio, per rendere le cose comprensibili, utilizzeremo questa grammatica LALR per le espressioni aritmetiche:

E \rightarrow E+T
E \rightarrow T
T \rightarrow T*F
T \rightarrow F
F \rightarrow id

Grammatica nello stile bottom-up
Non va bene per il top-down perché mostra ricorsione a sinistra (quindi non è LL-1)

Un possibile input è dato dalla stringa `3+4*5`

La fase di analisi lessicale produce una sequenza di token



A questo punto, comincia l'analisi sintattica

Quello che facciamo è considerare la stringa come `num+num*num` e capire se appartiene al linguaggio della grammatica.

Se l'analisi sintattica ha successo, dovremmo essere in grado di computare il valore dell'espressione.

Come dicevamo, possiamo fare questa operazione nello stesso momento in cui facciamo l'analisi sintattica. E come si fa?

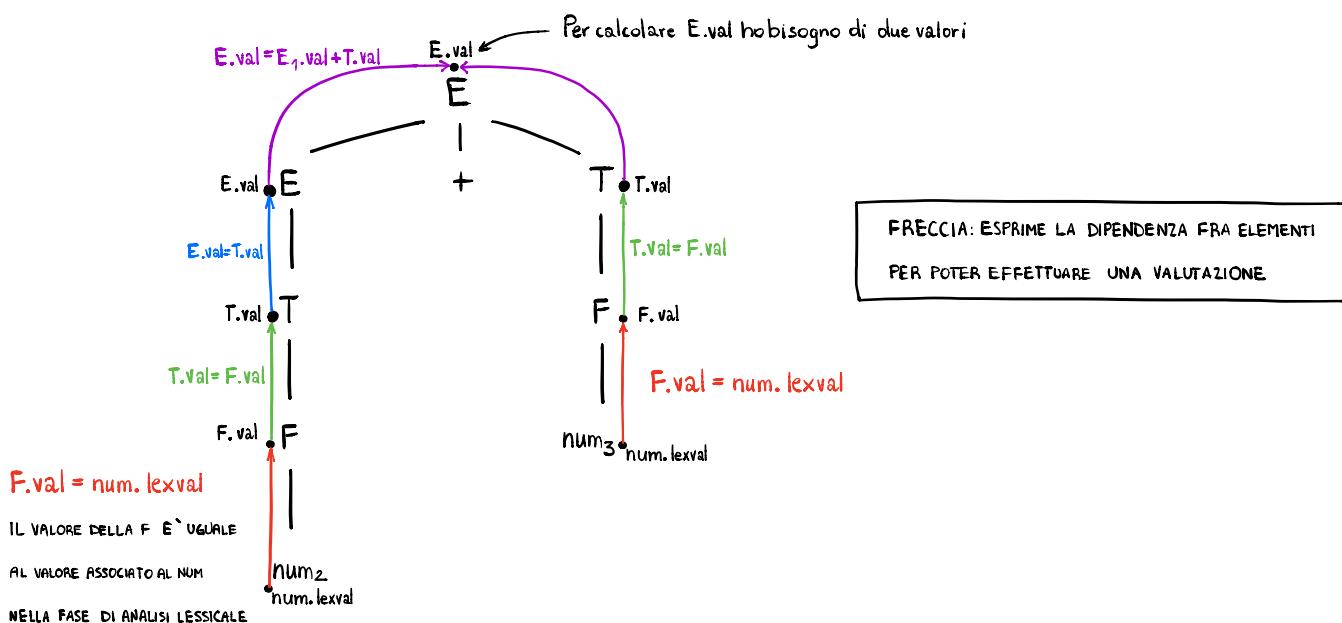
Vediamo intanto come si scrive

Si arricchisce la grammatica, facendola diventare una syntax-directed definition.

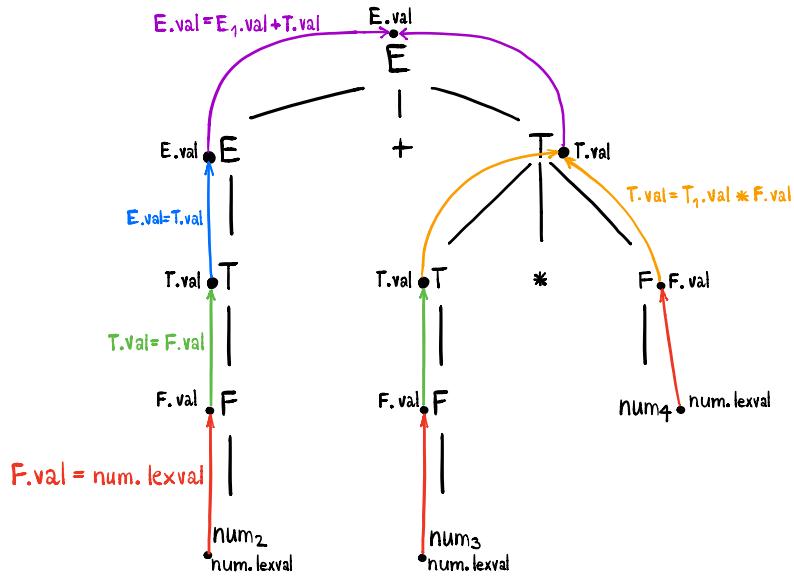
Si ammette la grammatica, lasciando diventare una syntax-directed definition:

$$\begin{array}{l} E \rightarrow E_1 + T \quad \{ E.\text{val} = E_1.\text{val} + T.\text{val} \} \\ E \rightarrow T \quad \{ E.\text{val} = T.\text{val} \} \\ T \rightarrow T_1 * F \quad \{ T.\text{val} = T_1.\text{val} * F.\text{val} \} \\ T \rightarrow F \quad \{ T.\text{val} = F.\text{val} \} \\ F \rightarrow \text{num} \quad \{ F.\text{val} = \text{num.lexval} \} \end{array} \quad \left\{ \begin{array}{l} \text{Abbiamo messo queste regole affinché ci consentano di valutare il valore} \\ \text{delle espressioni in ingresso.} \end{array} \right.$$

Per capire, vediamo subito un esempio: $2+3$. Disegnamp l'albero di derivazione in nero e l'albero annotato a colori:



Proviamo con un altro esempio: $2+3 \times 4$. Gli alberi sono:



Le dipendenze vanno, come dire, all'insù, e questo ci suggerisce che stiamo usando degli attributi **sintetizzati**, cioè l'attributo del padre (l'attributo del driver della produzione) è espresso in funzione degli attributi dei figli (gli attributi dei simboli che stanno nel body della produzione).

Dal momento che le dipendenze vanno all'insù, la computazione dell'espressione può essere fatta direttamente mentre si fa l'analisi sintattica.

In altre parole:

Quando effettuiamo una riduzione, sfruttiamo la regola associata alla produzione per la quale stiamo riducendo.

Se, in questo processo, i valori sono di volta in volta noti, nessuno vieta di arrivare in fondo con l'albero e anche il risultato.

Ciò detto, è necessario osservare che è molto importante che la grammatica e le regole siano ben organizzate.

Se una certa regola necessita di informazioni che ancora non sono disponibili, la computazione non può funzionare.

Il risultato in assoluto migliore che possiamo ottenere è organizzare grammatica e attributi in modo tale che riusciamo a fare tutto nella medesima passata (calcolo di albero e attributi).

Tipologie di attributi:

Attributi sintetizzati: per la produzione $A \rightarrow \alpha$, si dice sintetizzato l'attributo $A.at$ ($at = \text{attributo}$), definito come funzione di attributi dei simboli che occorrono in α .

Attributi ereditati: per la produzione $B \rightarrow \alpha A\beta$ si dice ereditato l'attributo $A.at$, definito come funzione degli attributi di B e dei simboli che occorrono in α e in β .

Siccome i terminali non hanno figli:

I simboli terminali della grammatica hanno solo attributi sintetizzati provenienti dall'analisi lessicale. Non possono esserci regole per computarli.

La grammatica che abbiamo visto prima aveva tutti attributi sintetizzati

La grammatica che abbiamo visto prima aveva tutti attributi sintetizzatori. La definizione è proprio "valore di padre dipende da valore di figlio".

Le grammatiche *s-attribuite*, cioè quelle in cui tutti gli attributi sono sintetizzati, sono ideali per l'analisi bottom-up. Visto che la direzione delle dipendenze fra attributi va dalle foglie alla radice, una qualunque visita in post-ordine dell'albero denotato fa computare il valore associato alla radice. (Nb: post-ordine = depth-first)

Vediamo ora una frazione della grammatica LL delle espressioni aritmetiche, in cui abbiamo eliminato la ricorsione a sinistra, le parentesi e il +.

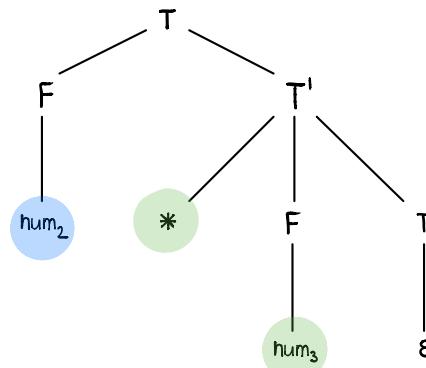
$T \rightarrow FT'$

$T' \rightarrow *FT' \quad \text{Grammatica nello stile bottom-up}$

$T' \rightarrow \epsilon$

$F \rightarrow \text{num}$

Sia l'input $2 * 3$. L'albero di derivazione è:



Questo albero ci mette in difficoltà, in quanto il $*$ e num_3 vengono da una parte, mentre num_2 viene dall'altra.

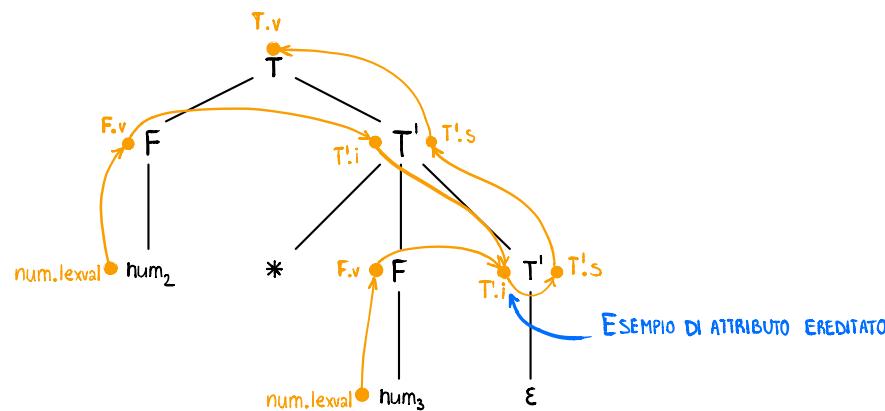
Qua vengono in soccorso gli **attributi ereditati**, che permettono di portare le informazioni di qua e di là nell'albero.

La grammatica arricchita è:

```

T → FT'      { T.v = T'.s, T'.i = F.v }
T' → *FT'    { T'.s = T'1.s, T'1.i = T'.i * F.v }
T' → ε        { T's. = T'.i }
F → num      { F.v = num.lexval }
  
```

L'albero di derivazione annotato è:



Gli attributi ereditati sono quelli espressi come funzione degli attributi del padre o dei fratelli.

Notiamo che, con la grammatica precedente, avevamo una struttura estremamente semplice, in cui bastava una visita dell'albero annotato in post-ordine.

Con quest'ultima grammatica, invece, una visita dell'albero annotato in post-ordine non è in grado di valutarlo. Quello che ci serve, in questo caso, è una individuare un ordinamento **topologico**.

Come si **imposta** il dependency graph?

- Per ogni nodo del parse tree e per ogni attributo del simbolo rappresentato da quel nodo, impostare un nodo del dependency graph.
- Per ogni attributo $A.a$ e per ogni attributo $X.x$ usato per definire $A.a$, impostare un arco tra i nodi che rappresentano le coppie corrispondenti di $X.x$ e $A.a$. L'arco va da $X.x$ a $A.a$ intendendo che $X.x$ è necessario alla computazione di $A.a$.

Come si **valuta** il dependency graph?

Numerare i nodi N_1, \dots, N_k in modo che, se c'è un arco da N_i ad N_j ($N_i \rightarrow N_j$), allora $i < j$.

Questo embedding è un sort topologico.

Lo SDD può essere valutato utilizzando un qualunque sort topologico, se esiste.

Osservazione:

Se c'è un ciclo nel dependency graph, allora non esiste alcun sort topologico dei suoi nodi, e lo SDD non può essere valutato.

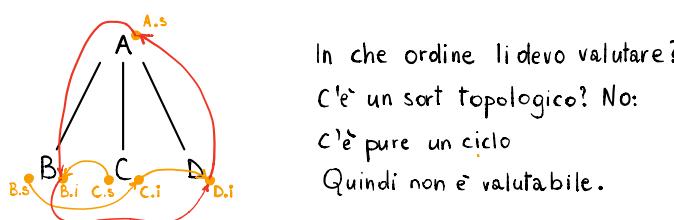
Ad esempio, supponiamo di avere una grammatica

con la seguente produzione: $A \rightarrow BCD$,

e con le seguenti regole semantiche associate alla produzione:

```
{   A.s = D.i  
    B.i = A.s + C.s  
    C.i = B.s  
    D.i = B.i + C.i }
```

Dato una stringa generabile da questa grammatica, quello che noi avremmo (in una qualche zona) del parse tree, sarebbe:



Però, noi vorremmo avere delle garanzie. Non è che tutte le volte che cerchiamo di mettere degli attributi alla grammatica, vogliamo *anche* chiederci se poi mai riusciremo a valutarla.

Ci sono delle classi specifiche di syntax-directed definition per le quali è dimostrato che si può sempre trovare un ordine topologico e quindi sono sempre valutabili.

Si tratta di due classi di grammatiche attribuite: S-attribuite ed L-attribuite.

SDD **S-attribuite**: tutti gli attributi utilizzati sono di tipo sintetizzato.

La valutazione dello SDD può essere conseguita con una visita in **post-ordine** del dependency graph.

Un esempio è $E \rightarrow E+T \mid T$.

In generale, tutte quelle le grammatiche LALR, o che comunque si prestano al bottom-up parsing, se sono tali per cui possiamo scrivere gli attributi stando attenti che siano sintetizzati, possono essere svolte con questa tecnica. Questo vuol dire che risuciamo a valutare il dependency graph mentre facciamo il parsing bottom up.

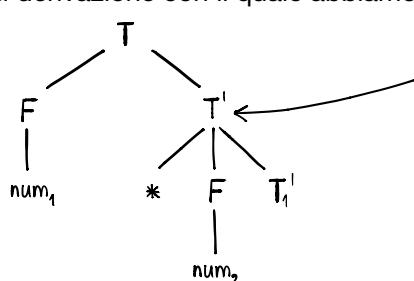
SDD **L-attribuite**: gli attributi utilizzati sono:

- sintetizzati, oppure
- ereditati, ma con il vincolo che per ogni produzione $A \rightarrow X_1 \dots X_n$ e per ogni $X_j.i$ (attributo ereditato di X_j) usa solamente attributi ereditati di A (può ereditare dal padre) e attributi ereditati o sintetizzati di $X_1 \dots X_{j-1}$ (attributi ereditati o sintetizzati dai fratelli a sinistra).

La grammatica per le espressioni aritmetiche di tipo LL che abbiamo visto ieri, è esattamente una SDD L-attribuita, in quanto ha esattamente questo tipo di schema. Rivediamola:

```
T → FT'  
T' → *FT'_1  
T' → ε  
F → num
```

Il problema che ci era sembrato difficile da gestire era questo: se prendiamo anche un semplice $num * num$, l'albero di derivazione con il quale abbiamo a che fare ha questa tipologia:



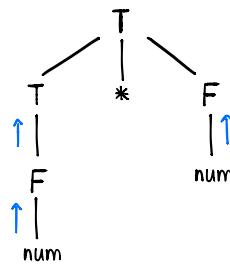
Nel momento in cui arriviamo in questa posizione, abbiamo conoscenza di num_2 (sappiamo che sarà un operando, un fattore, da coinvolgere nella moltiplicazione), ma ci manca completamente conoscenza dell'altro num .

Osserviamo che, con la grammatica di tipo bottom-up per lo stesso linguaggio, cioè

$T \rightarrow T^*F \mid F$

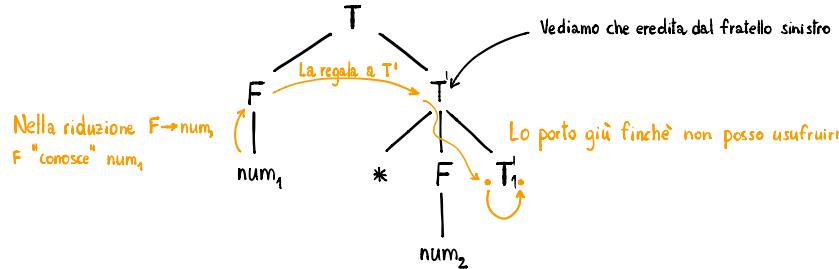
$F \rightarrow \text{num}$

L'albero che si ottiene per la moltiplicazione fra due numeri è:



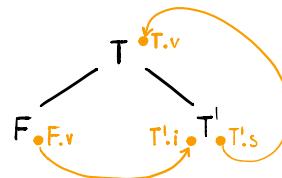
Qui, banalmente, si trasportano in su valori dei `num`, e quando si fa la riduzione, risulta anche possibile fare la moltiplicazione.

Quindi, abbiamo detto, la strategia che vogliamo utilizzare è di potarci il `num1` che abbiamo visto:

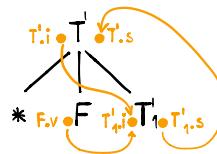


In particolare, vediamo cosa capita per le varie produzioni.

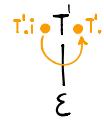
$T \rightarrow FT' \quad \{ \quad T'.i = F.v, \quad T.v = T.s \}$



$T' \rightarrow *FT'^1 \quad \{ \quad T'^1.i = T'.i * F.v, \quad T'.s = T'^1.s \}$



$T \rightarrow \varepsilon \quad \{ \quad T'.s = T'.i \}$



$F \rightarrow \text{num} \quad \{ \quad F.v = \text{num.lexval} \}$



Questa è una SDD L-attribuita. Tutti e quanti gli attributi ereditati che stiamo utilizzando sono attributi che utilizzano attributi del padre e attributi dei fratelli sinistri.

E infatti, per questa qui, l'ordine topologico lo troviamo senza problemi.

Trovare la SSD data questa grammatica:

D → TL
T → int
T → float
L → L1, id
L → id

Questo è lo spezzato della grammatica di un linguaggio di programmazione che permette di scrivere una cosa tipo:

int a, b

Per intendere che le variabili a, b sono di tipo intero.

Grazie all'analisi lessicale, la tabella dei simboli contiene una entry per la a, una entry per la b, ecc. Ricordiamo che nella tabella dei simboli mettiamo tutte e quante le informazioni che ci servono sui particolari elementi che stiamo considerando. Ad esempio, di una dichiarazione di procedura, ci interesserebbe sapere il numero e il tipo dei parametri.

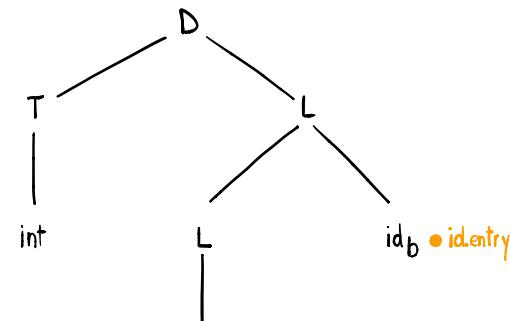
La tabella avrà questo aspetto:

TIPO	
a	int
b	int
c	int

In questo esercizio, immaginiamo di avere a disposizione una `insertType(id.entry, istanzaDiTipo)`, che possiamo usare successivamente come regola semantica.

Come organizzare una SSD per int a, b?

Ricordiamo che, quando usiamo `insertType`, ci serve sapere come si chiama l'identificatore, e che tipo ha. La storia è sempre quella: bisogna capire quando sono disponibili queste due informazioni.

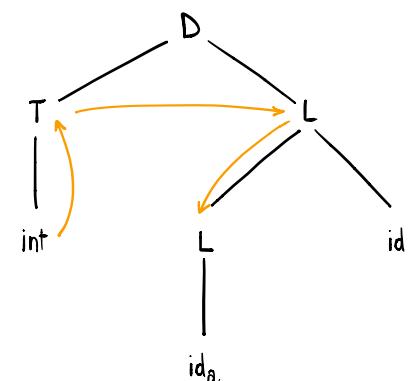


Attributo sintetizzato, analogo di $\xrightarrow{\quad}$ $\text{identry}_a \bullet \text{id}_a$
 num.lexval

Sta di fatto che, per invocare la `insertType` di questa cosa qua, dobbiamo aver visto sia l'int, che sta a sinistra, che id.entry, che stanno a destra, dall'altra parte. Dobbiamo attrezzarci con gli attributi.

Soluzione:

```
T → int { T.s = integer }
D → TL { L.i = T.s }
L → L1, id { L1.i = L.i, insertType(id.entry, L.i) }
L → id { insertType(id.entry, L.i) }
```

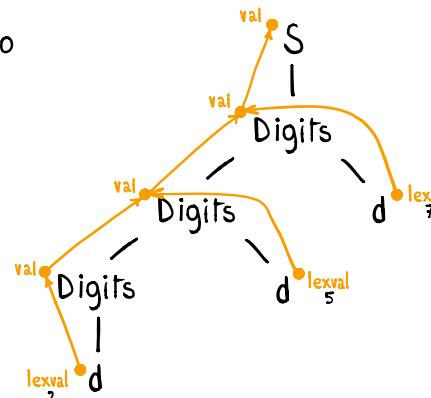


Esercizio 1

Da una stringa di caratteri che rappresentano cifre, ottenere il valore del numero.

```
S -> Digits           { S.val = Digits.val }
Digits -> Digits1 d  { Digits.val = Digits1.val * 10 + d.lexval }
Digits -> d          { Digits.val = d.lexval }
```

ESEMPIO

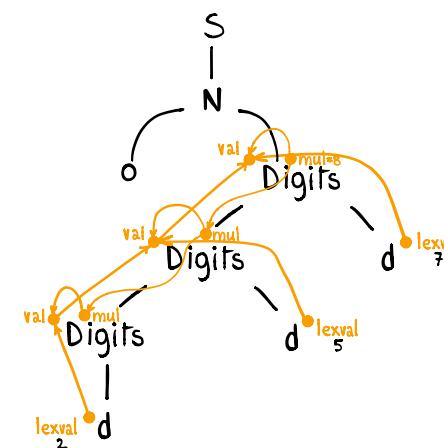


L'SDD è S-attribuito: solo attributi sintetizzati:
Quindi con un post-ordine si può valutare.

Esercizio 2

Come prima, solo che la stringa può (o no) cominciare con un prefisso 'o' che indica che i caratteri rappresentano cifre ottali anziché decimali.

```
S -> N
N -> o Digits      { S.val = N.val }
N -> Digits
Digits -> Digits1 d { N.val = Digits.val, Digits.mul = 8 }
Digits -> Digits1 d { N.val = Digits.val, Digits.mul = 10 }
Digits -> d          { Digits.val = Digits.mul * Digits1.val + d.lexval, Digits1.mul = Digits.mul }
Digits -> d          { Digits.val = d.lexval }
```



Esercizio 3

Un robottino, rappresentato da un punto, che parte dall'incrocio degli assi cartesiani, si può muovere in su, in giù, a sinistra e a destra. Data una stringa di caratteri che rappresentano gli spostamenti del robottino (WASD), restituire la posizione finale.

```
S -> Moves
Moves -> w Moves1   { Moves.y = Moves1.y +1, Moves.x = Moves1.x }
Moves -> s Moves1   { Moves.y = Moves1.y -1, Moves.x = Moves1.x }
Moves -> a Moves1   { Moves.y = Moves1.y, Moves.x = Moves1.x -1 }
Moves -> d Moves1   { Moves.y = Moves1.y, Moves.x = Moves1.x +1 }
Moves -> epsilon     { Moves.x = 0, Moves.y = 0 }
```

La lezione di oggi è consistita in un ripasso generale, ma sono state accennate alcune cose interessanti, che riporto in questo documento, senza un ordine particolare.

Il back-end del compilatore effettua anche ottimizzazioni machine-dependent:

Ad esempio, **common subexpression elimination**:

La seguente porzione di codice contiene la ripetizione dell'operazione $z * \pi$, che è onerosa:

```
x := z * pi * 2
```

...

```
y := z * pi * 5
```

Diventa:

```
a := z * pi
```

```
x := a * 2
```

...

```
y := a * 5
```

Ottimizzazione **definition use**: si basa su un grafo.

[https://en.m.wikipedia.org/wiki/Reaching_definition \[?\]](https://en.m.wikipedia.org/wiki/Reaching_definition)

Quando si scrive un compilatore, il goal è quello di avere il massimo di informazioni estraibili in fase di analisi sintattica, sia che questa sia top-down che bottom-up.

Ho due modi differenti per generare il codice intermedio:

Costruisco l'AST durante l'analisi sintattica → dall'AST genero il codice intermedio.

Arricchisco la grammatica con azioni semantiche, ottenendo un SDD → genero direttamente in fase di analisi sintattica.

Il più famoso three-address code si chiama SSA (Single State Assignment). Che agevola le operazioni di utilizzazioni

Pyhton e linguaggi di ultima generazione: l'indentazione ha un significato semantico, tipo quello della parentesi:

solo che la parentesi la vedo, la tabulazione no.

Accennato: scannerless parsing.

Grammatica:

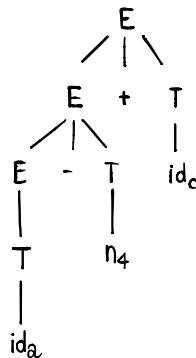
$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow (E) \mid id \mid n$

Stringa in input: $a-4+c$

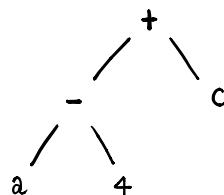
Riusciamo a trovare l'albero di derivazione? Sì: quindi la stringa in input appartiene al linguaggio della grammatica.

ALBERO DI DERIVAZIONE (O PARSE TREE)

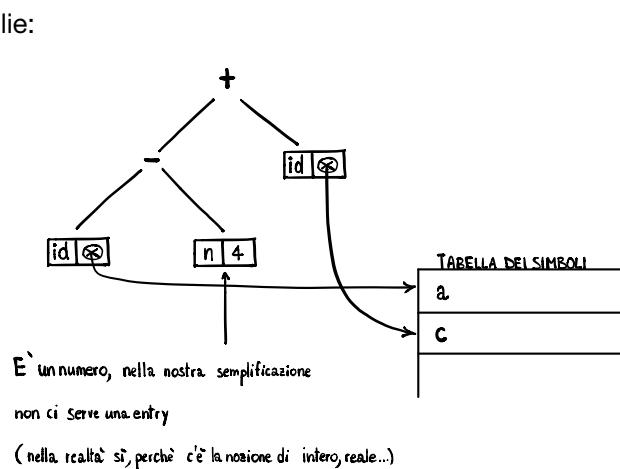


Ma l'utente vorrebbe l'abstract syntax tree, la cui struttura è molto più sintetica:

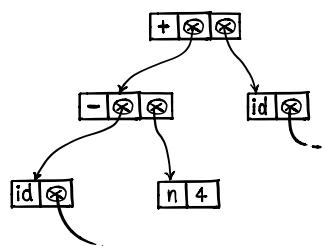
ABSTRACT SYNTAX TREE



Come rappresentiamo l'abstract syntax tree in memoria? In memoria avremo un record per ogni nodo (o foglia):



Vediamo i nodi intermedi:

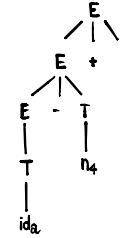


I nodi intermedi salvano la struttura topologica dell'albero

L'abstract syntax tree appena visto è una struttura che vogliamo generare mentre facciamo l'analisi sintattica. Possiamo conseguire ciò specificando delle regole semantiche (azioni).

Supponiamo di avere due funzioni che allocano lo spazio per generare un nodo o una foglia: new node (...) e new leaf (...)

```
T -> n      { T.node = new leaf (n, n.lexval) }
T -> id     { T.node = new leaf (id, id.entry) }
T -> (E)    { T.node = E.node }
E -> T      { E.node = T.node }
E -> E1-T   { E.node = new node ('-', E1.node, T.node) }
E -> E1+T   { E.node = new node ('+', E1.node, T.node) }
```



Supponiamo di avere in input la stringa $a - 4 + c$. Applichiamo lo shift-reduce algorithm (supponendo di avere già la tabella di parsing) e vediamo come questa grammatica attribuita ci permette di ottenere un parse tree.

In corrispondenza delle direttive di reduce vengono eseguite le regole semantiche. "Coloriamo" i puntatori (indirizzi):

Direttiva Effetto della regola semantica

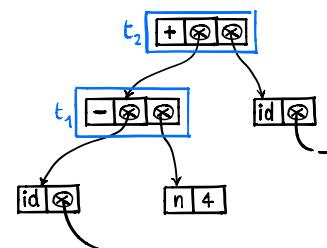
reduce T -> id_a		→ ENTRY NELLA TABELLA DEI SIMBOLI
reduce E -> T		E.node = ●
reduce T -> n4		n 4
reduce E -> E1-T		- ● ●
reduce T -> id_c		id → ENTRY
reduce E -> E1+T		+ ● ●

L'abstract syntax tree è importante, perché permette di fare molti controlli, che altrimenti dovrebbero essere fatti direttamente sul parse tree, ma mettendo delle regole semantiche più complesse.

Ora, dall'abstract syntax tree, è possibile generare del codice intermedio, che, in questo caso, sarebbe:

```
t1 = a - 4
t2 = t1 + c
```

Ci sono arrivato associando ad ogni nodo nodo intermedio un temporaneo:



Ma siamo sicuri che, per generare il codice intermedio, devo necessariamente passare prima dall'abstract syntax tree? No. È possibile conseguire l'ottenimento del codice intermedio aggiungendo alla grammatica degli attributi opportuni!

Supponiamo di avere la stessa grammatica di prima, con una produzione in più, $S \rightarrow id := E$, e con i seguenti attributi:

```
T -> n      { T.at = n.lexval } (dove at = attributo)
T -> id     { T.at = table.get(id.lexval) }
T -> (E)    { T.at = E.at }
E -> T      { E.at = T.at }
E -> E1-T   { E.at = new Temp(), generate(E.at '=' E1.at '-' T.at) }
E -> E1+T   { E.at = new Temp(), generate(E.at '=' E1.at '+' T.at) }
S -> id:=E { generate(table.get(id.lexval) '=' E.at) }
```

Supponiamo di avere in input la stringa $z := a - 4 + c$.

Applichiamo lo shift-reduce algorithm (supponendo di avere già la tabella di parsing).

In corrispondenza delle direttive di reduce vengono eseguite le regole semantiche.

Direttiva	Effetto della regola semantica
------------------	---------------------------------------

reduce T -> ida	T.at = a
-----------------	----------

reduce E -> T	E.at = a
---------------	----------

reduce T -> n4	T.at = 4
----------------	----------

reduce E -> E1-T	E.at = t ₁ , "t ₁ = a - 4"
------------------	--

reduce T -> idc	T.at = c
-----------------	----------

reduce E -> E1+T	E.at = t ₂ , "t ₂ = t ₁ + c"
------------------	---

reduce S -> id:=E	"z = t ₂ "
-------------------	-----------------------