Linguaggi Formali e Traduttori

2014 – 2015

2. Linguaggi regolari e automi a stati finiti

$a \rightarrow terminali$
$A \rightarrow non\ terminali$
$\alpha \rightarrow terminali\ o\ non\ terminali$

Operazioni sui linguaggi

- **Unione**: $L \cup M = \{w \mid w \in L \vee w \in M\}$
- **Intersezione:** $L \cap M = \{w \mid w \in L \wedge w \in M\}$
- **Concatenazione:** $L.M = \{w \mid w = xy \ \wedge \ x \in L \ \wedge \ y \in M\}$
- **Potenze**: $L^0 = \{\varepsilon\}$ $\quad L^1 = L$ $\quad L^{k+1} = L.L^k$
- **Chiusura di Kleene**: $L^* = \bigcup_{i=0}^{\infty} L^i$
- **Chiusura positiva di Kleene**: $L^+ = \bigcup_{i=1}^{\infty} L^i$

DFA – Automa deterministico a stati finiti

Un DFA è una quintupla:

$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q: insieme finito di stati
- Σ: alfabeto finito (= simboli in input)
- $\delta: Q \times \Sigma \rightarrow Q$ (funzione di transizione)
- $q_0 \in Q$: stato iniziale
- $F \subseteq Q$: insieme degli stati finali (anche vuoto)

NFA – Automa non deterministico a stati finiti

Un NFA è una quintupla:

$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q: insieme finito di stati
- Σ: alfabeto finito (= simboli in input)
- $\delta: Q \times \Sigma \rightarrow 2^Q$ (2^Q è il numero dei sottoinsiemi di Q)
- $q_0 \in Q$: stato iniziale
- $\mathrm{F} \subseteq Q$: insieme degli stati finali (anche vuoto)

Da NFA a DFA

Dato un NFA:

$$N = (Q_N, \Sigma, \delta_N, q_0, F_N)$$

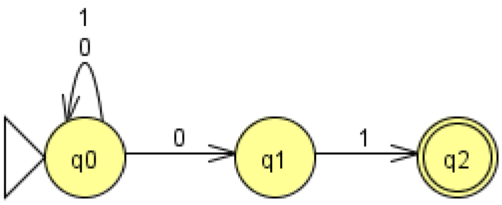
si costruisce un DFA:

$$D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$$

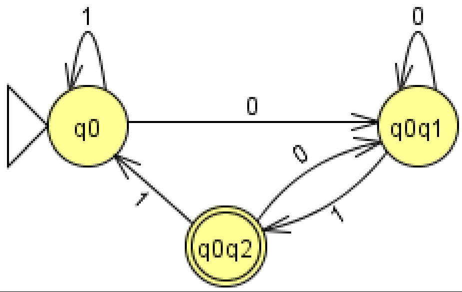
tale che $L(D) = L(N)$.

I dettagli della *costruzione a sottoinsiemi*:

- $Q_D = \{S \mid S \subseteq Q_N\}$
- $F_D = \{S \subseteq Q_N : S \cap F_N \neq \Phi\}$
- $\forall S \subseteq Q_N \wedge a \in \Sigma: \delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$



	0	1
Φ	Φ	Φ
$\rightarrow \{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$*\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$



1) Costruisci una tabella con tante colonne quanti sono i simboli dell'alfabeto
2) Inizi a riempire la tabella partendo dallo stato iniziale q_0
3) Ogni volta che trovi un nuovo insieme di stati, lo metti a sx e continui a segnare le transizioni

<u>STATO INIZIALE</u> = vecchio stato iniziale
<u>STATI FINALI</u> = stati che contengono almeno un vecchio stato finale

Teorema 2.11: Sia D il DFA ottenuto da un NFA N con la costruzione a sottoinsiemi. Allora L(D) = L(N).
Teorema 2.12: Un linguaggio L è accettato da un DFA sse L è accettato da un DFA.

ε-NFA – Automa non deterministico a stati finiti con ε-transizioni

Un ε-NFA è una quintupla:

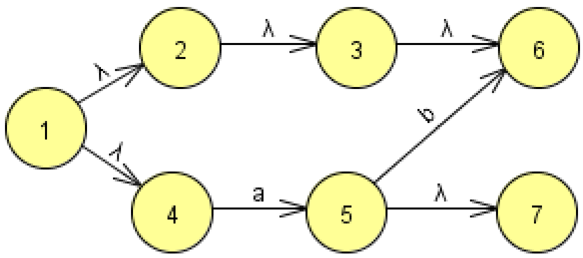
$$(Q, \Sigma, \delta, q_0, F)$$

- Q: insieme finito di stati
- Σ: alfabeto finito (= simboli in input)
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ (2^Q è il numero dei sottoinsiemi di Q)
- $q_0 \in Q$: stato iniziale
- $\mathrm{F} \subseteq Q$: insieme degli stati finali (anche vuoto)

ε-chiusura

La ε-chiusura di uno stato è un insieme di stati che contiene lo stato stesso e tutti gli stati da esso raggiungibili tramite una sequenza εε… ε.

<u>Definizione induttiva di ECLOSE(q):</u>
Base: $q \in ECLOSE(q)$
Induzione: $p \in ECLOSE(q) \wedge r \in \delta(p, \varepsilon) \Rightarrow r \in ECLOSE(q)$



ECLOSE(1) = {1, 2, 4, 3, 6}
ECLOSE(2) = {2, 3, 6}
ECLOSE(3) = {3, 6}
ECLOSE(4) = {4}
ECLOSE(5) = {5, 7}
ECLOSE(6) = {6}
ECLOSE(7) = {7}

Da ε-NFA a DFA

Dato un ε-NFA

$$E = (Q_E, \Sigma, \delta_E, q_0, F_E)$$

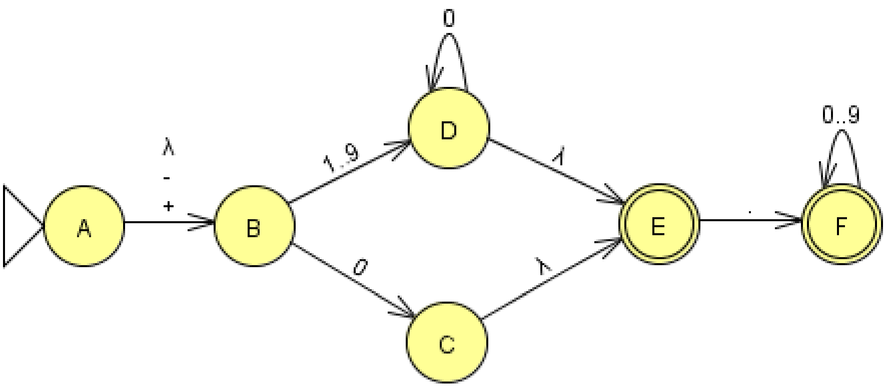
Si costruisce un DFA

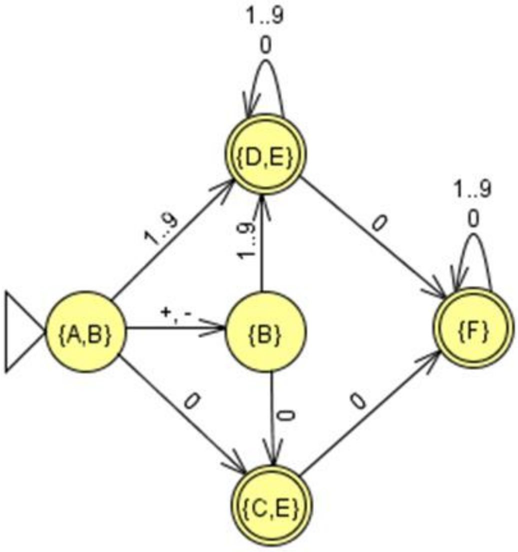
$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

tale che $L(D) = L(E)$.

Dettagli della costruzione:

- $Q_D = \{S : S \subseteq Q_E \wedge S = ECLOSE(S)\}$
- $q_D = ECLOSE(q_0)$
- $F_D = \{S \mid S \in Q_D \wedge S \cap F_E \neq \Phi\}$
- $\delta_D(S, a) = \cup\{ECLOSE(p) \mid p \in \cup_{t \in S} \delta(t, a)\}$





ECLOSE(A) = {A, B} **ECLOSE(C) = {C, E}** **ECLOSE(E) = {E}**
ECLOSE(B) = {B} **ECLOSE(D) = {D, E}** **ECLOSE(F) = {F}**

$$\boldsymbol{ECLOSE}\big(\boldsymbol{\delta}(\boldsymbol{A},+)\big) \cup \boldsymbol{ECLOSE}\big(\boldsymbol{\delta}(\boldsymbol{B},+)\big) = \{\boldsymbol{B}\}$$
$$\boldsymbol{ECLOSE}\big(\boldsymbol{\delta}(\boldsymbol{A},\mathbf{0})\big) \cup \boldsymbol{ECLOSE}\big(\boldsymbol{\delta}(\boldsymbol{B},\mathbf{0})\big) = \{\boldsymbol{C},\boldsymbol{E}\}$$

	+, -	0	1..9	.
→{A, B}	{B}	{C, E}	{D, E}	ϕ
{B}	ϕ	{C, E}	{D, E}	ϕ
*{C, E}	ϕ	ϕ	ϕ	{F}
*{D, E}	ϕ	{D, E}	{D, E}	{F}
*{F}	ϕ	{F}	{F}	ϕ

1) Trovo lo stato iniziale facendo ECLOSE(vecchio stato iniziale)
2) Per ogni stato nell'insieme di sx e per ogni simbolo in input nella tabella, trovo l'insieme di arrivo e di questo faccio l'ECLOSE

<u>STATO INIZIALE</u> = ECLOSE(vecchio stato iniziale)
<u>STATI FINALI</u> = stati che contengono almeno un vecchio stato finale

Teorema 2.22: Un linguaggio L è accettato da un ε -NFA E se e solo se L è accettato da un DFA.

3. Espressioni regolari

Espressioni regolari

Definizione induttiva:

Base:

- ε e ϕ sono espressioni regolari → L(ε) = {ε} e L(ϕ) = ϕ
- Se $a \in \Sigma$, allora a è un'espressione regolare → $L(a) = \{a\}$

Induzione:

- Se R è un'espressione regolare, allora (R) è un'espressione regolare → $L\big((R)\big) = L(R)$
- Se R e S sono espressioni regolari, allora R + S è un'espressione regolare → $\mathrm{L(R + S) \ = \ L(R) \cup L(S)}$
- Se R e S sono espressioni regolari, allora R.S (o S.R) è un'espressione regolare → $\mathrm{L(R.S) = \ L(R).L(S)}$
- Se R è un'espressione regolare, allora R* è un'espressione regolare → $L(R^*) = (L(R))^*$

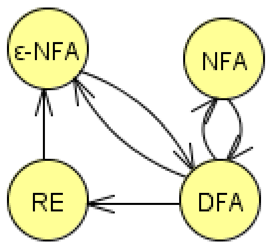
Ordine di precedenza: chiusura (*), concatenazione (.), unione (+)

Leggi algebriche per i linguaggi

	Linguaggi	**Espressioni regolari**
L'unione è commutativa	$L \cup M = M \cup L$	$R + S = S + R$
L'unione è associativa	$(L \cup M) \cup N = L \cup (M \cup N)$	$(R + S) + T = R + (S + T)$
La concatenazione è associativa	$(LM)N = L(MN)$	$(RS)T = R(ST)$
Φ è l'elemento neutro per l'unione	$L \cup \Phi = \Phi \cup L = L$	$R + \Phi = \Phi + R = R$
ε è l'elemento neutro sx e dx per la concatenazione	$\{\varepsilon\}L = L\{\varepsilon\} = L$	$\varepsilon R = R\varepsilon = R$
Φ è l'elemento zero sx e dx per la concatenazione	$\Phi \mathrm{L} = L\Phi = \Phi$	$\Phi \mathrm{R} = R\Phi = \Phi$
La concatenazione è distributiva a sx sull'unione	$L(M \cup N) = LM \cup LN$	$R(S + T) = RS + RT$
La concatenazione è distributiva a dx sull'unione	$(M \cup N)L = ML \cup NL$	$(S + T)R = SR + TR$
L'unione è idempotente	$L \cup L = L$	$R + R = R$
La chiusura è idempotente	$(L^*)^* = L^*$	$(R^*)^* = R^*$
	$\Phi^* = \{\varepsilon\}$	$\Phi^* = \varepsilon$
	$\{\varepsilon\}^* = \{\varepsilon\}$	$\varepsilon^* = \varepsilon$
	$LL^* = L^*L = L^+$	$RR^* = R^*R = R^+$
	$L^* = L^+ \cup \{\varepsilon\}$	$R^* = R^+ + \varepsilon$

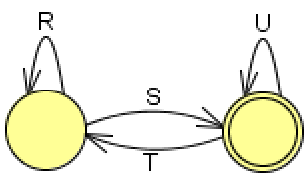
Abbiamo già mostrato che DFA, NFA, e ε-NFA sono tutti equivalenti.
Per mostrare che gli FA sono i riconoscitori dei linguaggi denotati dalle espressioni regolari, mostreremo che:

- Per ogni FA A possiamo costruire un'espressione regolare R, tale che L(R) = L(A)
- Per ogni espressione regolare R esiste un ε-NFA A, tale che L(A) = L(R).

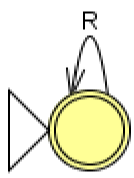


Tecnica di eliminazione degli stati

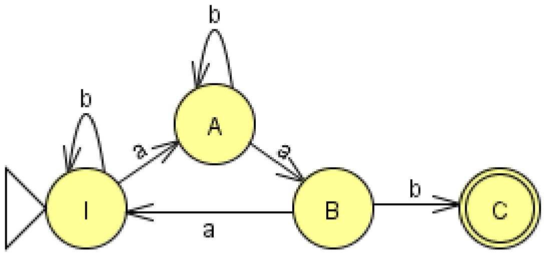
Eliminando uno stato per volta ed assegnando agli archi un'espressione regolare, dobbiamo arrivare in una di queste due situazioni:



$\boldsymbol{E_q = (R + SU^*T)^*SU^*}$

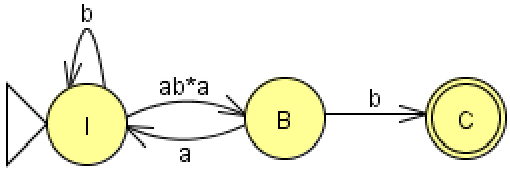


$\boldsymbol{E_q = R^*}$

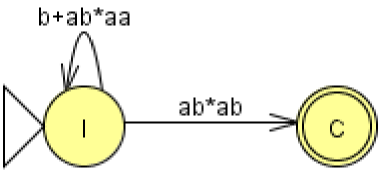


→

Elimino lo stato A:



Elimino lo stato B:



→ $E_q = ((b + ab^*aa) + ab^*ab)^*ab^*ab$

Sono arrivata alla prima forma $E_q = (R + SU^*T)^*SU^*$

Quindi:

→ **R = b + ab*aa**

S = ab*ab

U = ε

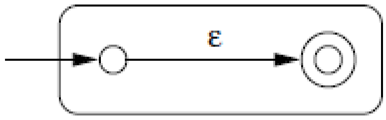
T = ε

Da espressione regolare a ε-NFA

Teorema 3.7: Per ogni espressione regolare R esiste un ε-NFA A, tale che L(A) = L(R).

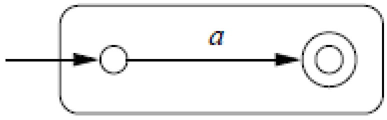
Costruzione per induzione strutturale:

Base: automa per ε, ϕ, a.



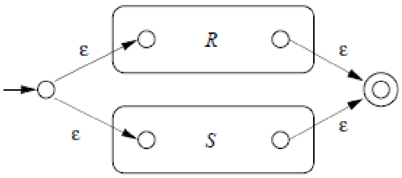
ε

ϕ

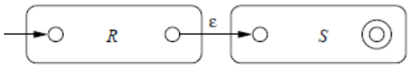


a

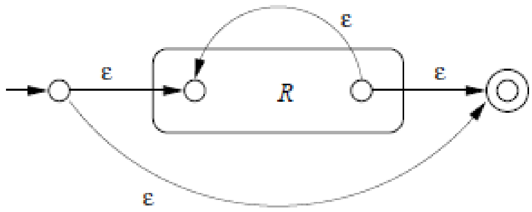
Induzione: automa per R+S, RS, R*



R+S



RS



R*

4. Proprietà dei linguaggi regolari

Pumping Lemma

Teorema 4.1: Sia L un linguaggio regolare. Allora $\exists n$, che dipende solo dal linguaggio, tale che $\forall\, w \in L, |w| \geq n$, w si può scrivere cine la concatenazione di tre sottostringhe xyz tali che:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. $\forall\, k \geq 0,\ xy^k z \in L$

Supponiamo che $L_{pr} = \{1^p \mid p \text{ è } un\ numero\ primo\}$ sia regolare.
Sia n la costante del pumping lemma.
Scegliamo un numero primo $p \geq n + 2$.

$$w = \overbrace{\underbrace{111\ldots}_{x}\underbrace{\ldots1}_{y}\ \underbrace{1111\ldots\ldots11}_{z}}^{p} \qquad |y| = m$$

Ora $xy^{p-m}z \in L_{pr}$.

$$|xy^{p-m}z| = |xy| + (p-m)|y| = p - m + (p-m)m = (1+m)(p-m)$$

che non è primo a meno che uno dei fattori non sia 1.

- $y \neq \varepsilon \Longrightarrow 1 + m > 1$
- $m = |y| \leq |xy| \leq n,\ p \geq n + 2 \Longrightarrow p - m \geq n + 2 - n = 2.$

Proprietà di chiusura dei linguaggi regolari

Siano L e M due linguaggi regolari. Allora i seguenti linguaggi sono regolari:

- Unione: L ∪ M
- Concatenazione: LM
- Chiusura di Kleene: L^*
- Complemento: $\overline{L}$
- Differenza: $L - M$
- Intersezione: $L \cap M$
- Inversione: $L^R = \{w^R \mid w \in L\}$

Chiusura rispetto all'unione L ∪ M

1) Aggiungi all'automa un nuovo stato inziale che si collega con ε-transizioni ai due vecchi stati iniziali
2) Ricopia i due automi vecchi

Chiusura rispetto al complemento $\overline{\mathbf{L}}$ (Vale sia per DFA che per NFA.)

3) Aggiungi all'automa lo stato pozzo (stato d'errore)
4) Rendi finali gli stati non finali e viceversa
5) Lo stato iniziale rimane lo stesso

Chiusura rispetto all'intersezione $L \cap M$

1) Costruisco la tabella partendo dallo stato composto dai due stati iniziali degli automi
2) Vado avanti con la tabella finché sulla sx ho stati "composti"
3) Gli stati finali sono quelli che contengono solo stati finali

Chiusura rispetto all'inversione $\mathbf{L^R}$ (Vale sia per DFA che per NFA.)

1) Giro tutti gli archi
2) Rendo il vecchio stato iniziale l'unico stato finale
3) Creiamo un nuovo stato iniziale p_0, con $\delta(p_0, \varepsilon) = F$ (gli stati finali del vecchio automa). Cioè:
   - Se nel vecchio automa c'è un unico stato finale → lo rendo iniziale (e non più finale)
   - Se nel vecchio automa ci sono più stati finali → creo un nuovo stato inziale e lo collego a tutti i vecchi stati finali con un ε-transizione.

Stati equivalenti

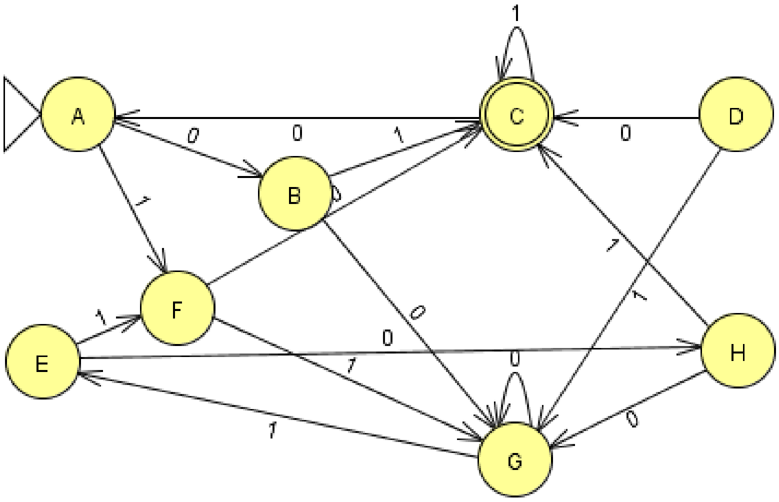
Sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA e $\{p, q\} \subseteq Q$. Definiamo:

$$p \equiv q \Leftrightarrow \forall w \in \Sigma^*: \hat{\delta}(p, w) \in F \text{ se e solo se } \hat{\delta}(q, w) \in F$$

- Se $p \equiv q$ diciamo che p e q sono **equivalenti**
- Se $p \not\equiv q$ diciamo che p e q sono **distinguibili**

In altre parole, p e q sono distinguibili se e solo se:

$$\exists w: \hat{\delta}(p, w) \in F \text{ e } \hat{\delta}(q, w) \notin F \text{, o viceversa.}$$



$\boldsymbol{\hat{\delta}(C, \varepsilon) \in F, \hat{\delta}(G, \varepsilon) \notin F \Longrightarrow C \not\equiv G}$

$\boldsymbol{\hat{\delta}(A, 01) = C \in F, \hat{\delta}(G, 01) = E \notin F \Longrightarrow A \not\equiv G}$

$\boldsymbol{\hat{\delta}(A, \varepsilon) = A \notin F, \hat{\delta}(E, \varepsilon) = E \notin F \wedge \hat{\delta}(A, 1) = F = \hat{\delta}(E, 1)}$

Quindi, $\forall$ stringa x, $\boldsymbol{\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x) = \hat{\delta}(F, x)}$

$\boldsymbol{\hat{\delta}(A, 00) = G = \hat{\delta}(E, 00)}$ $\boldsymbol{\hat{\delta}(A, 01) = C = \hat{\delta}(E, 01)}$

Conclusione $\boldsymbol{A \equiv E}$

Algoritmo riempi-tabella

Vale solo per DFA.

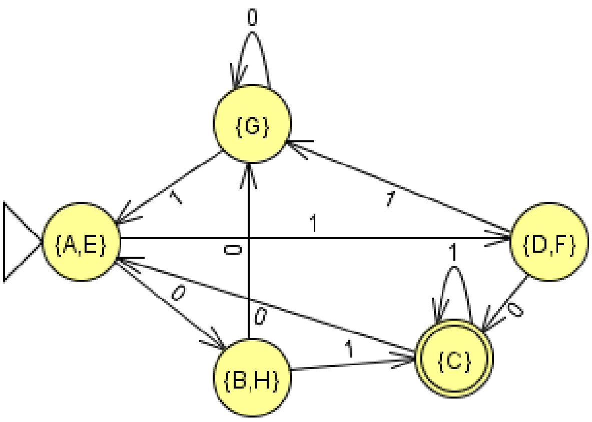
Possiamo calcolare coppie di stati distinguibili con il seguente metodo induttivo:

Base: Se $p \in F$ e $q \notin F$, allora $p \not\equiv q$

Induzione: Se $\exists a \in S: \delta(p, a) \not\equiv \delta(q, a)$, allora $p \not\equiv q$

	A	B	C	D	E	F	G
B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x

→ A ≡ E, B ≡ H, D ≡ F



Teorema 4.20: Se p e q sono stati di un DFA non distinti dall'algoritmo riempi-tabella, allora $p \equiv q$.

Teorema: Se p e q sono stati di un DFA e $p \equiv q$, allora p e q non sono distinti dall'algoritmo riempi-tabella.

$$\Downarrow$$

Teorema: Se p e q sono stati di un DFA, allora $p \equiv q$ se e solo se p e q non sono distinti dall'algoritmo riempi-tabella.

La relazione di equivalenza ($\equiv$) gode delle proprietà riflessiva, simmetrica e transitiva cioè è una relazione di equivalenza.

Teorema: Sia B il DFA ottenuto applicando l'algoritmo di minimizzazione al DFA A. Allora:

1. $L(B) = L(A)$.
2. L'automa B è minimo nel senso che non esiste nessun automa equivalente ad A con un numero di stati inferiore al numero di stati di B.
3. L'automa minimo è unico, a meno di isomorfismo (a meno del nome degli stati).

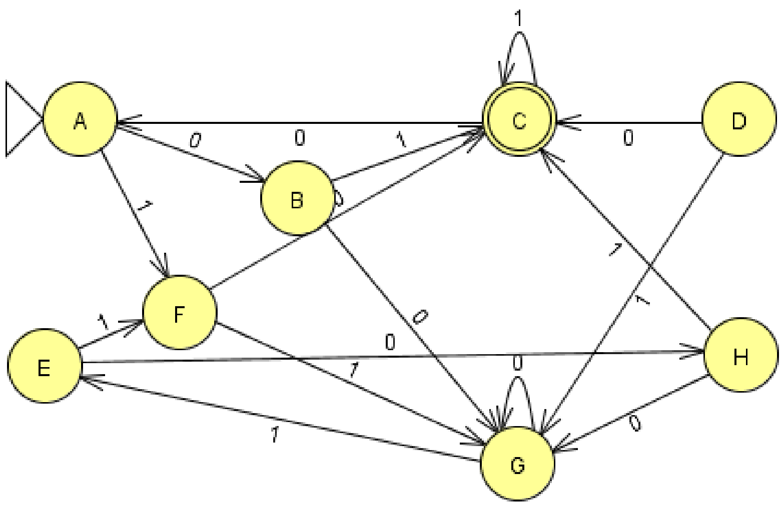
Un modo semplice per verificare se due automi finiti accettano lo stesso linguaggio è quello di minimizzarli. I due automi sono equivalenti se e solo se gli automi minimi ottenuti sono isomorfi.

Algoritmo di minimizzazione

Diciamo che una stringa x distingue lo stato s dallo stato t se esattamente uno degli stati raggiungibili da s e da t mediante un percorso etichettato con la stringa s è d'accettazione. Si dice inoltre che lo stato s è distinguibile t se esiste almeno una stringa che li distingue.

L'algoritmo di minimizzazione degli stati si basa sul partizionamento degli stati del DFA in gruppi di stati non distinguibili.
L'algoritmo modifica progressivamente una partizione i cui gruppi sono insiemi di stati non ancora identificati come distinguibili; due stati qualsiasi, appartenenti a insiemi diversi della partizione, sono invece già stati identificati come distinguibili.
Quando la partizione non può essere ulteriormente modificata spezzando un gruppo in gruppi più piccoli, allora essa rappresenta il DFA minimo.

Inizialmente la partizione contiene due gruppi di stati: gli stati finali e gli stati non finali.
Il procedimento fondamentale consiste nel considerare un generico gruppo $A = \{s_1, s_2, \ldots, s_k\}$ della partizione corrente e un generico simbolo dell'alfabeto a e verificare se il simbolo a può essere utilizzato per distinguere alcuni degli stati del gruppo A. A tale scopo si esaminano le transizioni da ognuno degli stati $s_1, s_2, \ldots, s_k$ relative al simbolo a; se gli stati raggiunti da tali transizioni ricadono in due o più gruppi della partizione corrente, si suddivide A in un insieme di gruppi in modo tale che due stati s_i e s_j siano nello stesso gruppo se e solo se, in corrispondenza del simbolo a, le transizioni da ognuno di essi portano a stati di uno stesso gruppo.
Si ripete questo procedimento finché nessun gruppo possa essere ulteriormente suddiviso per nessun simbolo di ingresso.



≡0 {A B D E F G H} {C}
≡1 {A B E G H} {C} {D F}
≡2 {A E} {G} {B H} {C} {D F}

5. Grammatiche libere dal contesto (CFG)

Una grammatica libera dal contesto è una quadrupla

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

- V: insieme finito di variabili
- T: insieme finito di terminali
- P è un insieme finito di produzioni della forma $A \to \alpha$ dove A è la variabile di testa e $\alpha \in (V \cup T)^*$ è il corpo
- S: è la variabile start symbol

I riconoscitori dei linguaggi context-free sono gli automi non deterministici.

Derivazione a sinistra e a destra

- **Derivazione a sinistra** $\Rightarrow_{lm}$: rimpiazza sempre la variabile più a sx con il corpo di una delle sue regole
- **Derivazione a destra** $\Rightarrow_{rm}$: rimpiazza sempre la variabile più a dx con il corpo di una delle sue regole

Linguaggio generato da una grammatica

Se $G = (V, T, P, S)$ è una CFG, allora il linguaggio G è:

$$L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$$

cioè l'insieme delle stringhe T^* derivabili dal simbolo iniziale.
Se G è una CFG, chiameremo $L(G)$ **linguaggio libero dal contesto**.

Forme sentenziali

- Sia $G = (V, T, P, S)$ una CFG, e $\alpha \in (V \cup T)^*$
- Se $S \Rightarrow^* \alpha$ diciamo che α è una **forma sentenziale**
- Se $S \Rightarrow_{lm} \alpha$ diciamo che α è una **forma sentenziale sinistra**
- Se $S \Rightarrow_{rm} \alpha$ diciamo che α è una **forma sentenziale destra**

$L(G)$ contiene le forme sentenziali che sono in T^*.

Alberi sintattici o di derivazione

Gli alberi sintattici o alberi di derivazione sono una rappresentazione delle derivazioni.
Se $w \in L(G)$, per una CFG, allora w ha un albero sintattico, che mostra la struttura sintattica di w
Ci possono essere diversi alberi sintattici per la stessa stringa.

Il **prodotto** di un albero sintattico è la stringa ottenuta leggendo le foglie da sinistra a destra.

- Ci interessano gli alberi sintattici in cui:
- Il prodotto è una stringa terminale.

La radice è etichettata dal simbolo iniziale.
L'insieme dei prodotti degli alberi sintattici è il linguaggio della grammatica.

In generale:

- Un albero sintattico descrive molte derivazioni, ma una sola derivazione sinistra e una sola derivazione a destra.
- Molte derivazioni **a sinistra** implica molti alberi sintattici.
- Molte derivazioni **a destra** implica molti alberi sintattici.

Teorema 5.29: Data una CFG G, una stringa terminale w ha due distinti alberi sintattici se e solo se w ha due distinte derivazioni a sinistra dal simbolo iniziale.

Sia $G = (V, T, P, S)$ una CFG e $A \in V$. Le seguenti affermazioni sono equivalenti:

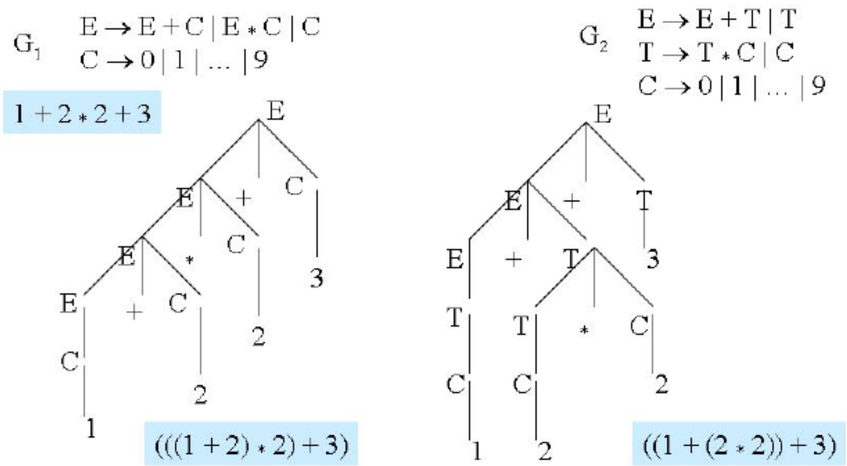
- $A \Rightarrow^* w$
- $A \Rightarrow_{lm}^* w$ e $A \Rightarrow_{rm}^* w$
- Vi è un albero sintattico di G con radice A e prodotto w

Ambiguità in grammatiche e linguaggi

L'esistenza di diverse derivazioni non è in generale un problema, costituisce un problema l'esistenza di diversi alberi sintattici.

Sia $G = (V, T, P, S)$ una CFG. Diciamo che G è **ambigua** se esiste una stringa in T^* che ha più di un albero sintattico.
Se ogni stringa in $L(G)$ ha al più un albero sintattico, G è detta **non ambigua**.

Ambiguità inerente: Un CFL L è inerentemente ambiguo se tutte le grammatiche per L sono ambigue.



Le due grammatiche sono non ambigue, ma solo la seconda è strutturalmente adeguata.

A volte possiamo rimuovere l'ambiguità ma non c'è nessun algoritmo per farlo in modo sistematico.
Alcuni CFL hanno solo CFG ambigue.

Eliminare l'ambiguità delle grammatiche

$$\boldsymbol{E \rightarrow I \mid E + E \mid E * E \mid (E) \mid num}$$
$$\boldsymbol{num \rightarrow 0 \mid 1 \mid \ldots \mid 9}$$

La grammatica a dx seguente è ambigua perché:

4. Non c'è precedenza tra $*$ e $+$
5. Non è specificato come interpretare le sequenze di uno stesso operatore:
   $\boldsymbol{E + E + E}$ è inteso come $\boldsymbol{E + (E + E)}$ o come $\boldsymbol{(E + E) + E}$?

Cerchiamo una grammatica non ambigua e strutturalmente adeguata equivalente a quella specificata per le espressioni parentesizzate che soddisfi i vincoli:

- **$*$ e $+$ associativi a sx**
- **$*$ ha precedenza su $+$**

Per il primo vincolo si deve eliminare una delle occorrenze di $\boldsymbol{E}$ nelle prime due produzioni. Ad esempio, se sostituiamo la produzione:

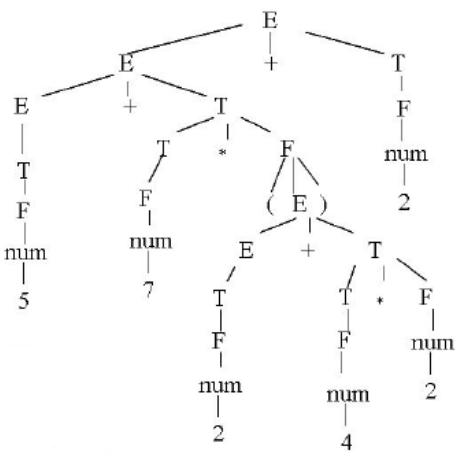
$$\boldsymbol{E \rightarrow E + E} \quad \textbf{con} \quad \boldsymbol{E \rightarrow E + num}$$

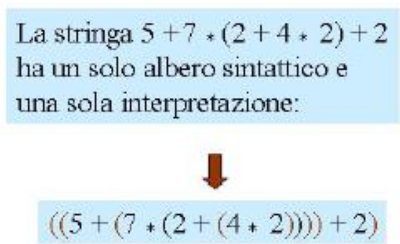
Si ottiene l'associatività a sinistra del $+$.

Per soddisfare i vincoli di precedenza si devono introdurre due variabili, una per ogni livello di precedenza e una variabile per le unità di base: $\boldsymbol{num}$ e espressioni tra parentesi.

Si ottiene quindi la seguente grammatica non ambigua:

$$\boldsymbol{E \rightarrow E + T \mid T}$$
$$\boldsymbol{T \rightarrow T * F \mid F}$$
$$\boldsymbol{F \rightarrow num \mid (E)}$$
$$\boldsymbol{num \rightarrow 0 \mid 1 \mid \ldots \mid 9}$$





6. Linguaggi regolari e linguaggi liberi dal contesto

Da espressione regolare a grammatica

Per induzione sulla struttura della espressione regolare:

- Se $E = a$, allora $S \rightarrow a$
- Se $E = \varepsilon$, allora $S \rightarrow \varepsilon$
- Se $E = F + G$, allora $S \rightarrow F \mid G$
- Se $E = FG$, allora $S \rightarrow FG$
- Se $E = F^*$, allora $S \rightarrow FS \mid \varepsilon$

Da automa a grammatica unilineare

- Un simbolo non-terminale per ogni stato
- Simbolo iniziale = stato iniziale
- Per ogni transizione da stato s a stato p con simbolo a, produzione $S \rightarrow aP$
- Se p stato finale, allora produzione $P \rightarrow \varepsilon$

Grammatiche dei linguaggi regolari

Una grammatica è unilineare destra se la sue regole hanno la forma:

$$X \rightarrow x\alpha \text{ dove } x \in \Sigma^* \text{ e } \alpha = A \in V \text{ o } \alpha = \varepsilon$$

$$\begin{aligned} &\boldsymbol{S \rightarrow A \mid B} \\ &\boldsymbol{A \rightarrow abA \mid \varepsilon} \\ &\boldsymbol{B \rightarrow bca} \end{aligned}$$

Dalle grammatiche unilineare destre alle espressioni regolari

Le regole di una grammatica unilineare destra (V, Σ, P, S) possono essere riscritte come $|V|$ equazioni in $|V|$ incognite (i non terminali delle grammatica) la cui soluzione è un'espressione regolare.

$$A \rightarrow \alpha_1 | \dots | \alpha_k \quad A \rightarrow \alpha_1 \cup \dots \cup \alpha_k$$

La soluzione si può calcolare iterando i seguenti passi:

1) Raccolta dei termini noti: $A = \left(u_{1,1} \cup u_{1,k_1}\right)B \cup \dots \cup (u_{h,1} \cup u_{h,k_h})B_h$
2) Eliminazione delle ricorsioni immediate: $A = sA \cup t \Longrightarrow A = (s)^*t$
3) Sostituzioni

$$\begin{aligned} &\boldsymbol{S \rightarrow A|B} \\ &\boldsymbol{A \rightarrow abA|\varepsilon} \\ &\boldsymbol{B \rightarrow bca} \end{aligned} \;\rightarrow\; \begin{aligned} &\boldsymbol{S = A \cup B} \\ &\boldsymbol{A = abA \cup \varepsilon} \\ &\boldsymbol{B = bca} \end{aligned} \;\rightarrow\; \begin{aligned} &\boldsymbol{S = A \cup B} \\ &\boldsymbol{A = (ab)^*\varepsilon} \\ &\boldsymbol{B = bca} \end{aligned} \;\rightarrow\; \boldsymbol{S = (ab)^* \cup bca}$$

Dalle grammatiche unilineare alle espressioni regolari

1. Se la variabile a sx non occorre nel membro destro, l'equazione resta inalterata. Altrimenti:
   - **Gramm. unilineare destra**: se l'eq. è ricorsiva $X = \alpha_1 X| \dots |\alpha_k X|\beta_1| \dots |\beta_h$, si raccolgono i coefficienti $X = (\alpha_1| \dots |\alpha_k)X|\beta_1| \dots |\beta_h$ e si risolve l'eq. eliminando la ricorsione:
   $$X = \alpha X|\beta \Longrightarrow X = \alpha^*\beta$$
   - **Gramm. unilineare sinistra**: se l'eq. è ricorsiva $X = X\alpha_1| \dots |X\alpha_k|\beta_1| \dots |\beta_h$, si raccolgono i coefficienti $X = X(\alpha_1| \dots |\alpha_k)|\beta_1| \dots |\beta_h$ e si risolve l'eq. eliminando la ricorsione:
   $$X = X\alpha|\beta \Longrightarrow X = \beta\alpha^*$$
2. Sostituire la variabile nelle altre equazioni e ricominciare dal passo 1. Fino a quando si ottiene un'equazione senza variabili.

$X = KX \cup L = K^*L$

$X = XK \cup L = LK^*$

7. Applicazioni delle CFG. Relazione con i linguaggi regolari

- Un linguaggio regolare è anche libero dal contesto.
- Da una espressione regolare, o da un automa, si può ottenere una grammatica libera dal contesto che genera lo stesso linguaggio.
- I linguaggi regolari sono però un sottoinsieme proprio dei linguaggi liberi e per ogni linguaggio regolare si può costruire una grammatica le cui produzioni hanno una forma particolare.

Grammatiche unilineari

Una grammatica è unilineare se le sue regole hanno la forma:

- Unilineare destra: $A \to uB \quad u \in \Sigma^*, B \in V \cup \{\varepsilon\}$
- Unilineare sinistra: $A \to Bv \quad v \in \Sigma^*, B \in V \cup \{\varepsilon\}$

In una grammatica unilineare non si possono mescolare produzioni del primo e del secondo tipo.
Le grammatiche unilineari generano i linguaggi regolari.

Proprietà: Ogni grammatica unilineare destra è equivalente a una grammatica unilineare sinistra, e viceversa.

Da CFG unilineare destra a CFG equivalente unilineare sinistra

Data una grammatica unilineare destra che genera il linguaggio L:

1. Costruire l'automa riconoscitore di L
2. Costruire l'automa riconoscitore di L^R
3. Trovare, a partire dall'automa, la grammatica G^R unilineare destra che genera L^R
4. Scrivere la grammatica G (unilineare sinistra) ottenuta da G^R rovesciando i membri destri delle produzioni.

Da CFG unilineare sinistra a CFG equivalente unilineare destra

Data una grammatica unilineare sinistra che genera il linguaggio L:

1. Scrivere la grammatica G^R (unilineare destra) ottenuta da G rovesciando i membri destri delle produzioni.
2. Costruire l'automa riconoscitore di L^R
3. Costruire l'automa riconoscitore di L
4. Trovare, a partire dall'automa, la grammatica G unilineare destra che genera L

Da grammatica unilineare destra a FA

Consideriamo grammatiche unilineari destre nelle cui produzioni siano presenti stringhe di terminali (u) di lunghezza al massimo uno.

- Uno stato per ogni variabile più uno stato finale.
- Stato iniziale = simbolo iniziale
- Per ogni produzione $A \to aB$ una transizione dallo stato A allo stato B etichettata con il simbolo a
- Per ogni produzione $A \to a$ una transizione dallo stato A allo stato finale etichettata con il simbolo a

8. Automi a pila (PDA)

Un automa a pila differisce da un automa finito, in particolare da un ε-NFA, per la presenza di una memoria a pila.
In una transizione, un PDA:

1) Consuma un simbolo in input o nessuno (mossa ε)
2) Il controllo transisce in un nuovo stato o resta nello stato in cui si trova
3) Sostituisce il simbolo sul top della pila con una stringa

Un PDA è una tupla di 7 elementi:

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

- Q: insieme finito di stati
- Σ: alfabeto finito di input
- Γ: alfabeto finito della pila
- $\delta: Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \to 2^{Q \times \Gamma^*}$ (funzione di transizione) δ(stato, input, top pila) = (stato, top pila) Pila: top..bottom
- q_0: stato iniziale
- $Z_0 \in \Gamma$: simbolo iniziale per la pila
- $F \subseteq Q$: insieme di stati finali

Gli automi a pila sono i riconoscitori dei linguaggi generati dalle grammatiche libere dal contesto, cioè dei linguaggi liberi dal contesto.

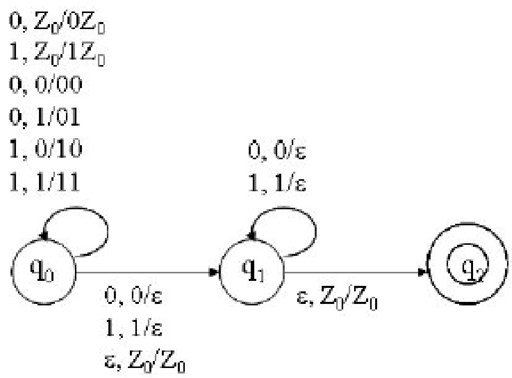
Un PDA per il linguaggio L_{ww^R} è la 7-tupla:

$$\boldsymbol{P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})}$$

dove δ è così definita:

$\boldsymbol{\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}}$
$\boldsymbol{\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}}$
$\boldsymbol{\delta(q_0, \varepsilon, Z_0) = \{(q_2, Z_0)\}}$
$\boldsymbol{\delta(q_0, 0, 0) = \{(q_0, 00), (q_1, \varepsilon)\}}$
$\boldsymbol{\delta(q_0, 1, 1) = \{(q_0, 11), (q_1, \varepsilon)\}}$
$\boldsymbol{\delta(q_0, 0, 1) = \{(q_0, 01)\}}$
$\boldsymbol{\delta(q_0, 1, 0) = \{(q_0, 10)\}}$

$\boldsymbol{\delta(q_1, 1, 1) = \{(q_1, \varepsilon)\}}$ **//tolgo il simbolo sul top**
$\boldsymbol{\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\}}$
$\boldsymbol{\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}}$



Accettazione per stato finale ($\mathbf{L = L(P)}$)

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Il linguaggio accettato da P per stato finale è:

$$L(P) = \{w | (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha), q \in F\}$$

- Si è raggiunto lo stato finale
- La stringa in input è stata tutta esaminata
- (non importa cosa resta sullo stack)

Il PDA visto accetta L_{ww^R} per stato finale.

Accettazione per pila vuota ($\mathbf{L = N(P)}$)

Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Il linguaggio accettato da P per pila vuota è:

$$N(P) = \{w | (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

dove q può essere uno stato qualunque.
Quando si indica un PDA che accetta per pila vuota si può omettere lo stato finale F.

- Lo stack è vuoto
- La stringa in input è stata tutta esaminata
- (non importa lo stato in cui mi trovo)

Da pila vuota a stato finale

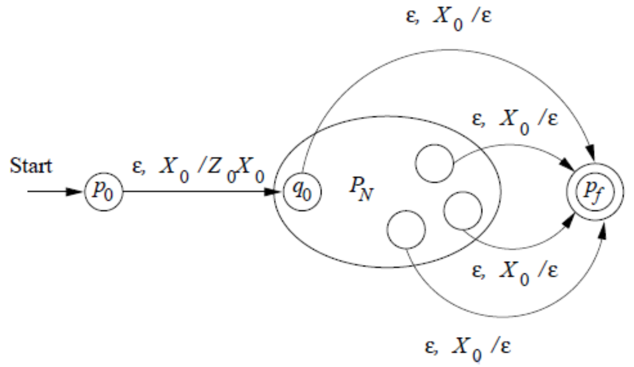
Teorema 6.9: Se $L = N(P_N)$ per un PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, allora ∃ PDA P_F tale che $L = L(P_F)$.

Costruzione:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

Dove:

- $\delta_F(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$ per un nuovo stato iniziale p_0
- per ogni $q \in Q, a \in \Sigma \cup \{\varepsilon\}, Y \in \Gamma$: $\delta_F(q, a, Y) = \delta_N(q, a, Y)$
- per ogni $q \in Q$, $(p_f, \varepsilon) \in \delta_F(q, \varepsilon, X_0)$ per un nuovo stato p_f accettante



Intuitivamente l'automa P_F inizia nel nuovo stato iniziale p_0:

1) Da p_0 poi si muove nello stato iniziale di q_0 di P_N impilando Z_0
2) In q_0 (trovando Z_0 su top dello stack) simula eseguendo le mosse di P_N, fino a quando non arriva alla pila vuota in un certo stato q
3) Da q, trovando X_0 sulla pila, si muove nel nuovo stato accettante p_f

Da stato finale a pila vuota

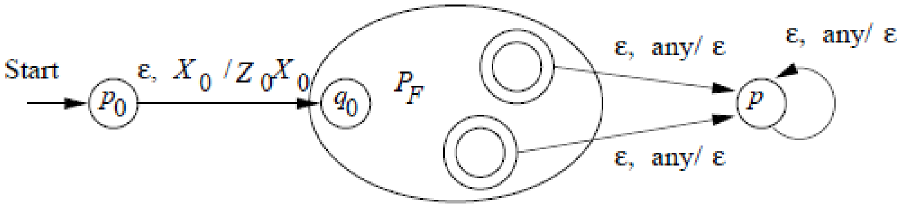
Teorema 6.11: Se $L = L(P_F)$ per un PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$, allora ∃ PDA P_N tale che $L = N(P_N)$.

Costruzione:

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

Dove:

- $\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$ per un nuovo stato iniziale p_0
- $\delta_N(p, \varepsilon, Y) = \{(p, \varepsilon)\}$, per un nuovo stato p per $Y \in \Gamma \cup \{X_0\}$
- per tutti i $q \in Q, a \in \Sigma \cup \{\varepsilon\}, Y \in \Gamma$: $\delta_N(q, a, Y) = \delta_F(q, a, Y)$
- $\forall q \in F, \wedge\, Y \in \Gamma \cup \{X_0\} : (p, \varepsilon) \in \delta_N(q, \varepsilon, Y)$.



Intuitivamente l'automa P_N inizia nel nuovo stato iniziale p_0:

1) Da p_0 poi si muove nello stato iniziale di q_0 di P_F impilando Z_0
2) In q_0 (trovando Z_0 su top dello stack) simula eseguendo le mosse di P_F, fino a quando non arriva in uno stato accettante $q \in F$
3) Da $q \in F$ si muove nel nuovo stato p, in cui la pila viene svuotata (senza leggere input)

Equivalenza di PDA e CFG

Un linguaggio è

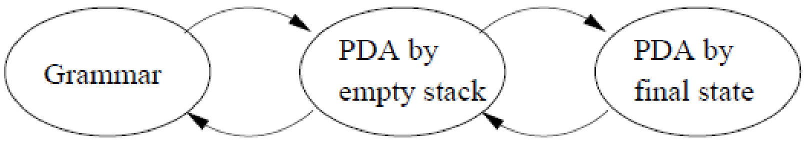
generato da una CFG

se e solo se è

accettato da un PDA per pila vuota

se e solo se è

accettato da un PDA per stato finale.



Da CFG a PDA

Data una grammatica G, costruiamo un PDA che accetta per pila vuota il linguaggio generato da G simulando una derivazione left-most.

Sia $G = (V, T, Q, S)$ una CFG. Definiamo P_G come

$$(\{q\}, T, V \cup T, \delta, q, S)$$

dove

$$\delta(q, \varepsilon, A) = \{(q, \beta) : A \rightarrow \beta \in Q\},$$

per ogni $A \in T$.

L'automa costruito dall'algoritmo:

- ha **un solo stato**
- accetta per **stack vuoto**
- è **non deterministico**

Quando c'è una corrispondenza tra il top e il carattere in input:

- sposto la testina
- elimino il carattere sul top (*es:* $\delta(q, a, a) = \{(q, \varepsilon)\}$)

Quando sul top c'è una variabile non terminale:

- sostituisco la variabile con tutti i membri destri di quella variabile

$S \rightarrow 0S1 \mid 0A1$
$A \rightarrow 2A \mid \varepsilon$
$\rightarrow$

- $\boldsymbol{\delta(q, \varepsilon, S) = \{(q, 0S1), (q, 0A1)\}}$ **→ la testina non avanza**
- $\boldsymbol{\delta(q, \varepsilon, A) = \{(q, 2A), (q, \varepsilon)\}}$ **→ la testina non avanza**
- $\boldsymbol{\delta(q, 0, 0) = \{(q, \varepsilon)\}}$ **→ la testina avanza**
- $\boldsymbol{\delta(q, 1, 1) = \{(q, \varepsilon)\}}$ **→ la testina avanza**
- $\boldsymbol{\delta(q, 2, 2) = \{(q, \varepsilon)\}}$ **→ la testina avanza**

1. Inserisci S sullo stack con una ε-mossa: $\boldsymbol{\delta(q, \varepsilon, Z_0) = (q, S)}$
2. ∀ non terminale A scrivi: $\boldsymbol{\delta(q, \varepsilon, A)} = \{(\boldsymbol{q}, \textbf{parte dx della prod}), (\boldsymbol{q}, \dots)\}$
3. ∀ terminale a scrivi: $\boldsymbol{\delta(q, a, a) = (q, \varepsilon)}$

Relazioni CFG-PDA

Teorema 6.13: Se P è il PDA costruito dalla CFG G nel modo descritto, allora $N(P) = L(G)$.

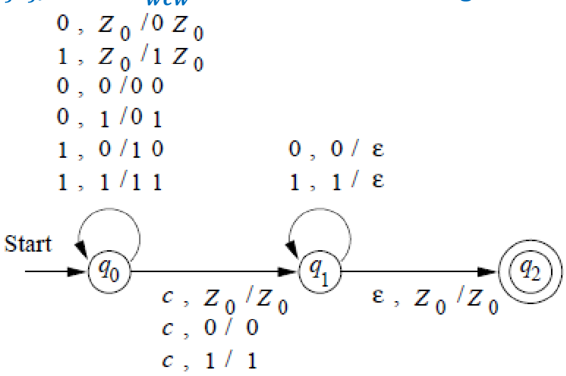
Teorema 6.14: Sia $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Allora esiste una CFG G tale che $L(G) = N(P)$.

PDA deterministici: DPDA

Un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ è **deterministico** se e solo se:

1. $|\delta(q, a, X)| \leq 1$ per tutte le terne (q, a, X)
2. $|\delta(q, \varepsilon, X)| \leq 1$ per tutte le coppie (q, X)
3. Se per ogni coppia (q, X) è definita una mossa spontanea, non è definita nessuna mossa con la lettura dallo stato q quando X è sul top dello stack (nessuna mossa su un simbolo di input).

Definiamo $\boldsymbol{L_{wcw^R} = \{wcw^R : w \in \{0,1\}^*\}}$, allora $\boldsymbol{L_{wcw^R}}$ è riconosciuto dal seguente DPDA:



Linguaggi regolari ⊆ L(DPDA)

Teorema 6.17: Se L è regolare, allora $L = L(P)$ (stato finale) per qualche DPDA P.

Costruzione: Dato che L è regolare, esiste un DFA A tale che $L = L(A)$. Sia:

$$A = (Q, \Sigma, \delta_A, q_0, F)$$

definiamo il DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_p, q_0, Z_0, F),$$

dove

$$\delta_p(q, a, Z_0) = \{(\delta_A(q, a), Z_0)\}$$

per tutti i $q \in Q$ e $a \in \Sigma$.

N(DPDA)

Proprietà del prefisso: Un linguaggio L ha la proprietà del prefisso se non esistono due stringhe distinte in L, tali che una è un prefisso dell'altra.

L_{wcw^R} ha la proprietà del prefisso mentre $\{0\}^*$.

I DPDA che accettano per pila vuota possono riconoscere solo CFL con la proprietà del prefisso.

Teorema 6.19: L è N(P) per qualche DPDA P se e solo se L ha la proprietà del prefisso e L è $L(P')$ per qualche DPDA P'.

Inclusioni

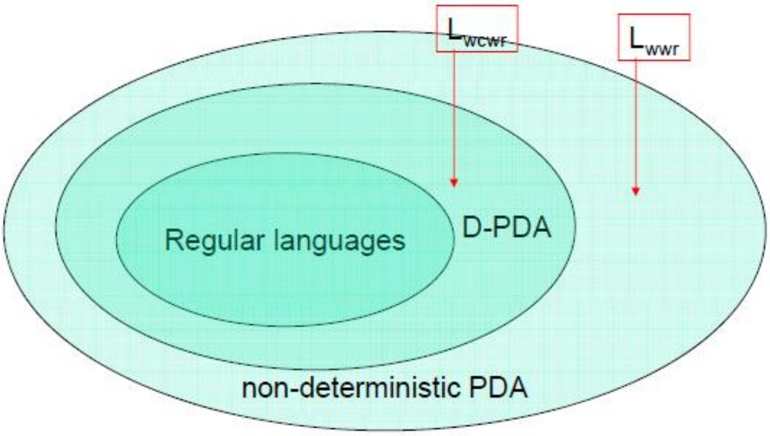
- $Linguaggi\ Regolari \subseteq L(DPDA)$

- $Linguaggi\ Regolari \subset L(DPDA)$
$L_{wcw^R} = \{wcw^R : w \in \{0,1\}^* \in L(DPDA) - Regolari$

- $L(DPDA) \subseteq CFL$

- $L(DPDA) \subset CFL$
$L_{ww^R} = \{ww^R : w \in \{0,1\}^* \in CFL - L(DPDA)$

- $Linguaggi\ Regolari \subset L(DPDA) \subset CFL$



DPDA e grammatiche non ambigue

Teoremi 6.20 e 6.21: Se L è un linguaggio accettato da un DPDA P per pila vuota ($L = N(P)$) o per stato finale ($L = L(P)$), allora L ha una CFG non ambigua.

Il viceversa non è vero: la classe dei linguaggi riconosciuti dai DPDA non coincide con quella dei linguaggi context-free non inerentemente ambigui.

L_{ww^R} ha una grammatica non ambigua $S \rightarrow 0S0 \mid 1S1\ \varepsilon$ ma non è nell'insieme L(DPDA).

9. Proprietà dei linguaggi liberi dal contesto

Semplificazione di grammatiche context-free

(1) Eliminare le ε-produzioni

Un non terminale è annullabile se produce la stringa vuota, cioè se $A \Longrightarrow^* \varepsilon$.

Data G, costruire G' equivalente a G, senza ε-produzioni:

1) Calcolare l'insieme $NULL$ dei non terminali annullabili
   $NULL_0 \ = \{A \mid A \rightarrow \varepsilon \in P\}$
   $NULL_i = NULL_{i-1} \cup \{A \mid A \rightarrow B_1 \dots B_n \in P \ \wedge \ \forall i(1 \leq i \leq n) B_i \in NULL_i\}$
   $NULL = NULL_n$ con n tale che $NULL_n = NULL_{n-1}$
2) Per ogni regola $A \rightarrow x_1 A_1 x_2 \dots x_n A_n x_{n+1} \in P$ aggiungere come regole alternative quelle ottenute cancellando in tutti i modi possibili dalla parte destra i non terminali A_i che sono annullabili
3) Eliminare le regole $A \rightarrow \varepsilon$
4) La grammatica ottenuta genera $L(G) - \{\varepsilon\}$, perciò se $L(G)$ contiene la stringa vuota (cioè S è annullabile), aggiungere $S \rightarrow \varepsilon$.

$S \rightarrow ABC$ $A \rightarrow EAF \mid a \mid EF$ $B \rightarrow b$ $C \rightarrow Sc \mid c$ $E \rightarrow \varepsilon$ $F \rightarrow d \mid \varepsilon$	$NULL = \{E, F, A\}$ →	$S \rightarrow ABC \mid BC$ $A \rightarrow EAF \mid a \mid EF \mid EA \mid$ $EF \mid AF \mid E \mid A \mid F$ $B \rightarrow b$ $C \rightarrow Sc \mid c$ $F \rightarrow d \mid \varepsilon$	E non è più definita →	$S \rightarrow ABC \mid BC$ $A \rightarrow a \mid AF \mid F$ $B \rightarrow b$ $C \rightarrow Sc \mid c$ $F \rightarrow d$

Se B $\Rightarrow^* \varepsilon$ e hai:

- $\boldsymbol{A \rightarrow BB}$**, allora** $\boldsymbol{S \rightarrow BB|B}$
- $\boldsymbol{A \rightarrow C}$**, allora rimane** $\boldsymbol{A \rightarrow C}$

(2) Eliminare le produzioni unitarie

Sostituire ogni produzione unitaria $A \rightarrow B$ con l'insieme delle produzioni $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$ dove $B \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k \in P$ (cioè sostituire B con tutte le sue riscritture) fino a quando non si hanno più produzioni unitarie. Ad ogni passo eliminare, qualora si presentino, le regole $A \rightarrow A$ che riscrivono una variabile con se stessa.

$S \rightarrow SbS \mid A$ $A \rightarrow a$	→	$S \rightarrow SbS \mid a$	$E \rightarrow E + T \mid T$ $T \rightarrow TxC \mid C$ $C \rightarrow a \mid b \mid c$	→	$E \rightarrow E + T \mid TxC \mid a \mid b \mid c$ $T \rightarrow TxC \mid a \mid b \mid c$ $C \rightarrow a \mid b \mid c$

(3) Eliminare i simboli inutili

1) Eliminare i non generatori (quelli da cui non si raggiunge uno stato finale)
   1) Trovare l'insieme GEN dei simboli che producono stringhe di terminali: $A \Longrightarrow^* w$
   2) Eliminare le produzioni che contengono le variabili in $V - GEN$
2) Eliminare i non raggiungibili (quelli non raggiungibili dallo stato iniziale)
   1) Cercare l'insieme RAG dei simboli presenti nelle derivazioni da S: $S \Rightarrow^* xAy$
   2) Eliminare le produzioni che contengono le variabili in $V - RAG$

$S \rightarrow AB \mid CaD$ $A \rightarrow aA \mid B$ $B \rightarrow bB$ $C \rightarrow Cb \mid a$ $D \rightarrow bD \mid a$	$GEN \ = \ \{C, D, S\}$ →	$S \rightarrow CaD$ $C \rightarrow Cb \mid a$ $D \rightarrow bD \mid a$
$S \rightarrow AA \mid aD$ $A \rightarrow aA \mid B$ $B \rightarrow bB \mid b$ $C \rightarrow Cb \mid aSb \mid b$ $D \rightarrow bD \mid a$	$RAG \ = \ \{S, A, D, B\}$ →	$S \rightarrow AA \mid aD$ $A \rightarrow aA \mid B$ $B \rightarrow bB \mid b$ $D \rightarrow bD \mid a$

Pumping lemma

Sia L un CFL. Esiste una costante k tale che, se $z \in L$ e $|z| \geq k$, possiamo scrivere $z = uvwxy$ con le seguenti condizioni:

1. $|vwx| \leq k$
2. $vx \neq \varepsilon$
3. $\forall i \geq 0, uv^i wx^i y \in L$

$L = \{0^n 1^n 2^n \mid n \geq 1\}$

Supponiamo L libero dal contesto; chiamiamo k la costante del pumping lemma.

Scegliamo $z = 0^k 1^k 2^k$.

Comunque noi spezziamo z in $uvwxy$, con $|vwx| \leq k$ e v e x non entrambe vuote, vwx non può contenere sia 0 che 1 e 2 in quanto l'ultimo 0 e il primo 2 sono lontani $n+1$ posizioni.

Ci sono i seguenti casi da considerare:

- **vwx è formata da un solo simbolo, ad esempio 0 (o 1 o 2). In tal caso anche vx è formata da un solo simbolo e uwy non ha un numero uguale di 0, 1 e 2.**
- **vwx non contiene 2. Allora vx ha solo 0 e 1. Quindi, uwy che dovrebbe essere in L, ha k 2, ma meno di k 0 e/o 1.**
- **vwx non contiene 0. Analogo al caso precedente.**

Limiti delle CFG

Le grammatiche libere dal contesto:

6. Non sono in grado di generare stringhe con tre o più gruppi di caratteri ripetuti lo stesso numero di volte.
7. Non sanno generare coppie con lo stesso numero di simboli, se le coppie sono "intrecciate".
   $L = \{0^i 0^j 0^i 0^j \mid i, j \geq 1\}$
   Sia k la costante del pumping lemma, scegliamo $z = 0^k 0^k 0^k 0^k$. Quindi vwx contiene un solo simbolo o al massimo due simboli. In ogni caso, le stringhe generate non sono in L.
8. Non sanno ripetere una stringa di lunghezza arbitraria, se la stringa è su un alfabeto di cardinalità maggiore di 1.
   $L = \{ww \mid w \in \{0,1\}^*\}$
   Sia k la costante del pumping lemma, scegliamo $z = 0^k 1^k 0^k 1^k$. Comunque la scomponiamo un $uvwxy$, è facile verificare che, ad esempio uv^2wx^2y non è una stringa di L.

Proprietà di chiusura dei linguaggi liberi dal contesto

Siano L e M due linguaggi regolari. Allora i seguenti linguaggi sono liberi dal contesto:

- Unione: $L \cup M$
- Concatenazione: LM
- Chiusura di Kleene: L^*
- Chiusura positiva: L^+

Siano $G_1 = < V_1, \Sigma_1, P_1, S_1 >$ e $G_2 = < V_2, \Sigma_2, P_2, S_2 >$ due grammatiche libere con insiemi di non terminali disgiunti (condizione che si può sempre ottenere) e sia S un nome "fresh":

- Una grammatica per l'**unione** è: $G_\cup = < V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}, S >$
- Una grammatica per la **concatenazione** è: $G_C = < V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S >$
- Una grammatica per la **chiusura di Kleene** è:
  $G_* = < V_1 \cup \{S\}, \Sigma_1, P_1 \cup \{S \to SS_1 \mid \varepsilon\}, S >$ oppure $G_* = < V_1 \cup \{S\}, \Sigma_1, P_1 \cup \{S \to S_1 S \mid \varepsilon\}, S >$
- Una grammatica per la **chiusura di Kleene positiva** è: $G_+ = < V_1 \cup \{S\}, \Sigma_1, P_1 \cup \{S \to SS_1 \mid S_1\}, S >$

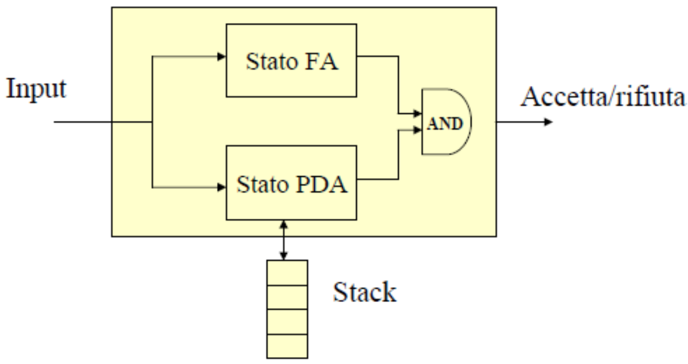
Teorema 7.25: Se L è CF, allora lo è anche L^R.

Supponiamo che L sia generato da $G = (V, T, P, S)$. Costruiamo $G^R = (V, T, P^R, S)$, dove $P^R = \{A \to \alpha^R : A \to \alpha \in P\}$, cioè inverti (rifletti) la parte destra delle produzioni.

Teorema 7.27: Se L è CF e R è regolare, allora $L \cap R$ è CF.

Sia L accettato dal PDA $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$ per stato finale, e sia R accettato dal DFA $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$

Costruiamo un PDA per $L \cap R$ secondo la figura:



Definiamo $P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta(q_0, q_A), Z_0, F_P \times F_A)$ dove

$\delta((q,p), a, X) = \left\{\left(\left(r, \hat{\delta}_A(p,a)\right), \gamma\right) \,\middle|\, (r,\gamma) \in \delta_P(q,a,X)\right\}.$

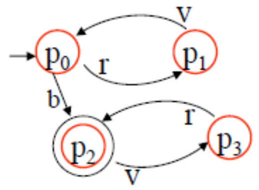
Nota: $a \in \Sigma \cup \{\varepsilon\}$.

$L_1 = \{wbw^R | w \in \{r,v\}^+\}$ e $L_2 = L((rv)^* b(vr)^*)$ **costruiamo l'automa che riconosce** $L_1 \cup L_2$.

Siano $P = (Q_1, \Sigma, \Gamma, \delta_1, q_0, Z_0, \{q_2\})$ **l'automa a pila che accetta** L_1:

$\delta_1(q_0, v, Z_0) = \{(q_0, vZ_0)\}$ $\delta_1(q_0, r, v) = \{(q_0, rv)\}$
$\delta_1(q_0, r, Z_0) = \{(q_0, rZ_0)\}$ $\delta_1(q_0, b, r/v) = \{(q_0, r/v)\}$
$\delta_1(q_0, v, v) = \{(q_0, vv)\}$ $\delta(q_1, r, r) = \{(q_1, \varepsilon)\}$
$\delta_1(q_0, r, r) = \{(q_0, rr)\}$ $\delta(q_1, v, r) = \{(q_1, \varepsilon)\}$
$\delta_1(q_0, v, r) = \{(q_0, vr)\}$ $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$

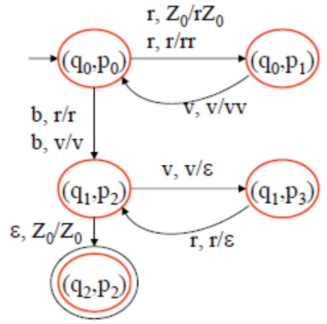
E $A = (Q_2, \Sigma, \delta_2, p_0, \{p_2\})$ **l'automa a stati finiti che accetta** L_2.



L'automa P' **che riconosce** $L_1 \cap L_2$ **è così definito:** $P' = (Q_1 \times Q_2, \Sigma, \Gamma, \delta(q_0, p_0), Z_0, \{(q_2, p_2)\})$

$\delta((q_0, p_0), r, Z_0) = \{(q_0, p_1), rZ_0)\}$
$\delta((q_0, p_0), b, r/v) = \{(q_1, p_2), r/v)\}$
$\delta((q_0, p_1), v, v) = \{(q_0, p_0), vv)\}$
$\delta((q_0, p_0), r, r) = \{(q_0, p_1), rr)\}$
$\delta((q_1, p_2), v, v) = \{(q_1, p_3), \varepsilon)\}$
$\delta((q_1, p_3), r, r) = \{(q_1, p_2), \varepsilon)\}$
$\delta((q_1, p_2), \varepsilon, Z_0) = \{(q_2, p_2), Z_0)\}$

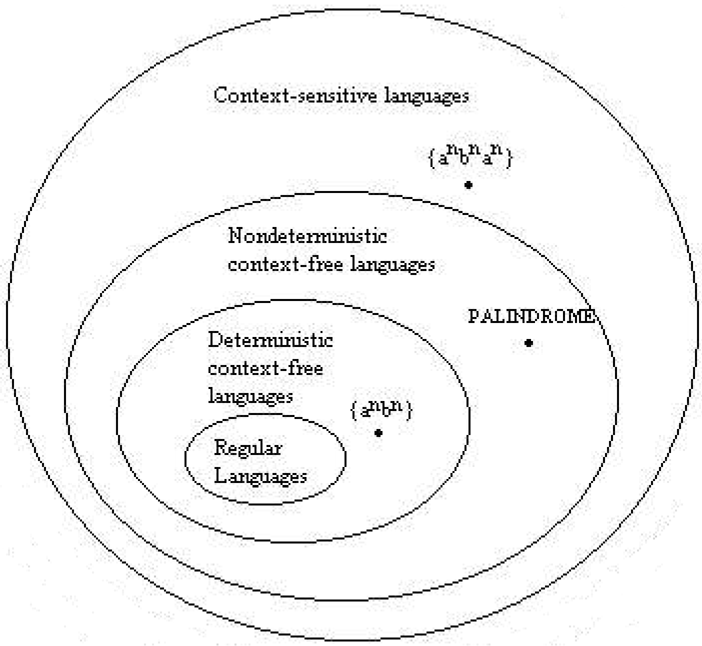
Nota: molti stati sono inutili e molte transizioni non sono definite, ad esempio $\delta\big((q_0, p_0), v, Z_0\big)$ **non è definita in quanto non è definita la transizione** $\delta_2(p_0, v)$.
Basta che una delle due transizioni non sia definita a rendere inutile la transizione.



Teorema 7.29: Siano L, L_1, L_2 CFL e R un linguaggio regolare. Allora:

- $L - R$ è CF
  (Se R è regolare, allora $L \cap \bar{R}$ è regolare, e $L \cap \bar{R} = L - R$)
- $\bar{L}$ non è necessariamente CF
  (Se $\bar{L}$ fosse sempre CF, seguirebbe che $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ sarebbe sempre CF)
- $L_1 - L_2$ non è necessariamente CF
  (Notare che Σ^* è CF, quindi se $L_1 - L_2$ fosse sempre CD, allora lo sarebbe sempre anche $\Sigma^* - L = \bar{L}$)

Classificazione di Chomsky



10. Parsificazione top-down e bottom-up

Due classi importanti di analizzatori:

Top-down (discendente)

- *Antlr* (generatore automatico di parser)
- Costruiscono un albero di parsificazione iniziando dalla radice e creando i nodi in preordine
- Ricerca di una derivazione **leftmost** per la stringa in input: espande le parti sinistre delle regole della grammatica nelle corrispondenti parti destre

Bottom-up (ascendente)

- *Yacc* (generatore automatico di parser)
- Costruiscono l'albero dalle foglie alla radice
- Ricerca di una derivazione **rightmost** della stringa in ordine contrario: riduce le parti destre delle regole della grammatica nei corrispondenti non terminali

La costruzione generale dell'automa a pila (non deterministico) sostituisce ad una variabile A in cima alla pila la parte destra di una produzione per A con una ε-mossa (quindi "alla cieca"). Se invece, per fare questa mossa, permettiamo di utilizzare k caratteri dell'input ancora da analizzare possiamo in certi casi scegliere in modo univoco la produzione da sostituire ad A sulla pila (cioè la produzione da usare in una derivazione a sinistra).

Il modello su cui si basano tutti gli analizzatori sintattici (più o meno fedelmente) è l'automa a pila.

LL(k)

- *La prima **L** indica che la stringa in input viene analizzata a partire da sinistra.*
  *La seconda **L** indica che si costruisce una derivazione sinistra.*
- Le grammatiche che permettono analisi sintattica top-down deterministica o "parsing predittivo discendente" sono chiamate $LL(k)$ (k è il numero di simboli necessari per individuare la produzione senza ambiguità).
- Radice → foglie.
- Un linguaggio è $LL(k)$ se esiste una grammatica $LL(k)$ che lo genera.
- La famiglia $LL(k)$ contiene tutti e soli i linguaggi che possono essere definiti da una grammatica $LL(k)$ per un valore finito di $k \geq 1$.
  Non tutti i linguaggi che hanno riconoscitori deterministici sono generabili da grammatiche $LL(k)$, cioè la famiglia dei linguaggi $LL(k)$ è strettamente contenuta nella famiglia dei linguaggi che hanno riconoscitori deterministici.

$L = \{a^n b^m \mid n \geq m > 0\}$ ammette un riconoscitore deterministico, ma non è $LL(k)$ per nessun k.

$S \to AB$	**Non si sa cosa fare sull'input a;**
$A \to aA \mid a$	**qualunque sia il numero di**
$B \to aBb \mid ab$	**simboli di look ahead.**

Ci sono stringhe che hanno un numero di a tale per cui non si è comunque in grado di decidere quale riscrittura di A usare.

LR(k)

- *La prima **L** indica che la stringa in input viene analizzata a partire da sinistra.*
  *La **R** indica che si costruisce una derivazione destra in ordine inverso.*
- Le grammatiche che permettono analisi sintattica bottom-up deterministica o "parsing predittivo ascendente" sono chiamate $LR(k)$
- Foglie → radice.
- Un linguaggio è $LR(k)$ se esiste una grammatica $LR(k)$ che lo genera.
- Più potente del top-down.
- Se gli elementi disponibili non permettono di prendere una decisione, una tattica efficace è quella di rinviarla, tenendo aperte diverse strade fino al momento in cui emergeranno situazioni nuove che permetteranno di decidere.
  Nell'intervallo tra la comparsa dell'incertezza e la sua risoluzione, il calcolo deve conservare le informazioni intermedie necessarie per eseguire la scelta in un secondo momento.
  Questa è l'idea alla base degli algoritmi di analisi ascendente $LR(k)$, che costruiscono un automa a pila dotato di più stati interni. Nel caso LR ha senso anche ridurre a 0 la lunghezza della prospezione, poiché nei casi più semplici la scelta può essere determinata solo in base allo stato.
- **La famiglia dei linguaggi accettati per stato finale da un automa push-down deterministico coincide con quella dei linguaggi generati dalle grammatiche $LR(1)$.**

11. Analisi discendente. Parsificazione LL: LL(1)

L'obiettivo è costruire un analizzatore deterministico, basato sul modello dell'automa a pila, che riconosca il linguaggio scegliendo in modo non ambiguo le produzioni da usare.
Esamineremo il caso in cui sia possibile scegliere deterministicamente la produzione da usare ad ogni passo esaminando solo un simbolo di input. Possiamo chiamarlo automa con "look ahead" di lunghezza 1.
<u>La scelta importante è determinare la produzione da applicare quando un non terminale si trova in cima alla pila.</u>
<u>Quando invece in cima alla pila si trova un terminale basterà verificare la corrispondenza di questo col simbolo in input e avanzare la testina di lettura.</u>
In mancanza di questa corrispondenza la stringa in input viene rifiutata.

FIRST (associati alle stringhe)

E' l'insieme dei terminali con cui iniziano le stringhe derivabili da α nella grammatica G: $F(\alpha) = \{a \mid \alpha \Longrightarrow^* a\beta\}$.
$F(\alpha)$ soddisfa la definizione ricorsiva:

1. $F(\varepsilon) = \{\varepsilon\}$
2. $F(a\beta) = \{a\}$
3. $F(A\beta) = \begin{cases} F(A) & se\ A\ non\ è\ annullabile \\ (F(A) - \{\varepsilon\}) \cup F(\beta) & se\ A\ è\ annullabile \end{cases}$ dove $F(A) = \bigcup_{A \to \gamma_1 \in P} F(\gamma_1)$

Algoritmo

Data $G = (V, \Sigma, P, S)$.
1. $F_n(a) \to \{a\}$ per ogni a in Σ e per ogni n
2. Trovare l'insieme $NULL$ delle variabili annullabili
3. Per tutte le variabili A in V, $F_0(A) \leftarrow \Phi$ se A non è annullabile e $F_0(A) \leftarrow \{\varepsilon\}$ se A è annullabile
4. $n \leftarrow 1$
   a) Per tutte le variabili A in V
      i. $j \leftarrow 1$
      ii. $F_n(A) \leftarrow F_n(A) \cup (F_{n-1}(Y_j) - \{\varepsilon\})$
      iii. $se\ Y_j$ è annullabile e $j < k, j \leftarrow j + 1$, tornare al punto ii
   b) $n \leftarrow n + 1$; ripetere dal passo a), a meno che, $\forall A \in V, F_n(A) = F_{n-1}(A)$
5. Per tutte le variabili A in V, $F(A) \leftarrow F_n(A)$

Cioè:
1) Trovo l'insieme delle variabili annullabili $NULL$
2) Inizializzo gli F_0 delle variabili $\in NULL$ a $\{\varepsilon\}$, gli altri a Φ
3)
4)

$\boldsymbol{Z \to d \mid XYZ}$		$\boldsymbol{F_0(Z) = \Phi}$	$\boldsymbol{F_0(X) = \{\varepsilon\}}$	$\boldsymbol{F_0(Y) = \{\varepsilon\}}$
$\boldsymbol{Y \to c \mid \varepsilon}$	$\boldsymbol{NULL = \{X, Y\}}$	$\boldsymbol{F_1(Z) = \{d\}}$	$\boldsymbol{F_1(X) = \{a, \varepsilon\}}$	$\boldsymbol{F_1(Y) = \{c, \varepsilon\}}$
$\boldsymbol{X \to Y \mid a}$	$\boldsymbol{\to}$	$\boldsymbol{F_2(Z) = \{d, a, c\}}$	$\boldsymbol{F_2(X) = \{a, c, \varepsilon\}}$	$\boldsymbol{F_2(Y) = \{c, \varepsilon\}}$
		$\boldsymbol{F_3(Z) = \{d, a, c\}}$	$\boldsymbol{F_3(X) = \{a, c, \varepsilon\}}$	$\boldsymbol{F_3(Y) = \{c, \varepsilon\}}$

FOLLOW (associati alle variabili)

E' l'insieme dei terminali con cui iniziano le stringhe che seguono A nelle derivazioni della grammatica G: $FW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta\}$.
$FW(A)$ soddisfa la seguente definizione:

$$FW(A) = \left(\bigcup_{B \to \alpha A \beta \in P} (F(\beta) - \{\varepsilon\}) \right) \cup \left(\bigcup_{\substack{B \to \alpha A \beta \in P\ tali\ che \\ \beta\ annullabile\ e\ B \neq A}} FW(B) \right) \cup$$
$$\cup \{\$\}\ se\ A\ è\ lo\ start\ symbol$$

Algoritmo

Data $G = (V, \Sigma, P, S)$.

1. $\forall A \in V$
   a) Calcolare $F(A)$
   b) $FW_0(A) = \{F(\beta) - \{\varepsilon\} \mid B \to \alpha A \beta \in P\}$ e aggiungere \$ in $FW_0(S)$
2. $n \leftarrow 1$
   a) $\forall A \in V$
      i. $FW_n(A) \leftarrow FW_{n-1}(A)$
      ii. Per ogni produzione $B \to \alpha A \beta \in P$ tale che β è annullabile e $B \neq A$, $FW_n(A) \leftarrow FW_n(A) \cup FW_{n-1}(B)$
   b) $n \leftarrow n + 1$; ripetere dal passo a), a meno che $\forall A \in V, FW_n(A) = FW_{n-1}(A)$
3. $\forall A \in V, FW(A) \leftarrow FW_n(A)$

Cioè:

1) Li cerchi dei membri destri delle produzioni e sono gli inizi della variabile successiva oppure il terminale successivo
2) $FW_0(start\ symbol) = \{\$ \cup \dots\}$
3) Se $K \to N$ e cerchi $FW(N)$ allora prendi gli $FW(K)$ perché dopo N non c'è nulla
4) Negli FW non ci sono mai gli ε

$$\begin{array}{l} Z \to d \mid XYZ \\ Y \to c \mid \varepsilon \\ X \to Y \mid a \end{array} \quad \rightarrow \quad \begin{array}{lll} FW_0(Z) = \{\$\} & FW_0(X) = \{a, c, d\} & FW_0(Y) = \{a, c, d\} \\ FW_1(Z) = \{\$\} & FW_1(X) = \{a, c, d\} & FW_0(Y) = \{a, c, d\} \end{array}$$

Insiemi guida (associato alle produzioni)

Data una grammatica G, l'insieme guida di una produzione $A \to \alpha$ è l'insieme dei terminali con cui iniziano le stringhe generate da A usando la produzione.

$Gui(A \to \alpha)$ è così definito:

$$Gui(A \to \alpha) = \begin{cases} F(\alpha) & se\ \alpha\ non\ è\ annullabile \\ (F(\alpha) - \{\varepsilon\}) \cup FW(A) & se\ \alpha\ è\ annullabile \end{cases}$$

$$\begin{aligned} Gui(Z \to d) &= F(d) = \{d\} \\ Gui(Z \to XYZ) &= F(XYZ) \\ &= (F(X) - \{\varepsilon\}) \cup F(YZ) \\ &= (F(X) - \{\varepsilon\}) \cup (F(Y) - \{\varepsilon\}) \cup F(Z) \\ &= \{d\} \end{aligned}$$

$$\begin{aligned} Gui(Y \to c) &= F(c) = \{c\} \\ Gui(Y \to \varepsilon) &= (F(\varepsilon) - \{\varepsilon\}) \cup FW(Y) = \{a, c, d\} \\ Gui(X \to Y) &= \{a, c, d\} \\ Gui(X \to a) &= \{a\} \end{aligned}$$

Grammatiche $LL(1)$

Una grammatica è $LL(1)$ se per ogni non terminale A e per ogni coppia di produzioni $A \to \alpha$ e $A \to \beta$, gli insiemi guida sono disgiunti:

$$Gui(A \to \alpha) \cap Gui(A \to \beta) = \Phi$$

Tabella di parsificazione LL(1)

	a	c	d	ε
Z			$Z \to d, Z \to XYZ$	
Y	$Y \to \varepsilon$	$Y \to c, Y \to \varepsilon$	$Y \to \varepsilon$	
X	$X \to Y, X \to a$	$X \to Y$	$X \to Y$	

Non è $LL(1)$

Automa LL(1)

Costruiamo a partire da una grammatica G un automa P con un solo stato, che accetta per stack vuoto, e con la capacità di leggere un simbolo in input senza spostare la testina.

$$P = (\{q\}, \Sigma, \Sigma \cup V, \delta, q, S)$$

$\delta(q, a, X) = (q, \beta) \implies$ se $a \in Gui(X \to \beta)$ (senza spostamento della testina)

$\delta(q, a, a) = (q, \varepsilon) \implies$ (con spostamento della testina)

Input	Stack	Derivazione
0011$	S	S
0011$	0A	0A
011$	A	
011$	S1	0S1
011$	0A1	00A1
11$	A1	
11$	11	0011
1$	1	
$		

Grammatica	Gui
$S \to 0A$	{0}
$A \to S1$	{0}
$A \to 1$	{1}

$if(Gui(STACK) == cc)$
  pop
  $push$(parte dx della produzione)
$if(STACK == cc)$
  pop
  $cc \leftarrow PROSS$

Modifichiamo pertanto la definizione della funzione di transizione rispetto a quella usata nella costruzione dell'automa a partire dalla grammatica, in modo da tener conto del primo carattere del membro destro delle produzioni:

- $\delta(q, a, X) = (q, \alpha)$ → Se $a \in Gui(X \to \alpha)$ e $a \neq a\beta$ (senza spostamento della testina)
- $\delta(q, a, X) = (q, \beta)$ → Se $a \in Gui(X \to \alpha)$ e $a \equiv a\beta$ (con spostamento della testina)
- $\delta(q, a, a) = (q, \varepsilon)$ → con spostamento della testina

Analizzatore a discesa (iterativo)

Produzione	Gui	
$S \to PQ$	$\{a, b, c\}$	$F(S) = \{a, b, c\}$
$Q \to \&PQ$	$\{\&\}$	$F(Q) = \{\varepsilon, \&\}$
$Q \to \varepsilon$	$\{\$\}$	$F(P) = \{a, b, c\}$
$P \to aPb$	$\{a\}$	$FW(S) = \{\$\}$
$P \to bPa$	$\{b\}$	$FW(Q) = \{\$\}$
$P \to c$	$\{c\}$	$FW(P) = \{a, b, \&, \$\}$

	a	b	c	&	$
S	$S \to PQ$	$S \to PQ$	$S \to PQ$		
Q				$Q \to \&PQ$	$Q \to \varepsilon$
P	$P \to aPb$	$P \to bPa$	$P \to c$		

```
Program esempio
  begin
    cc ←PROSS
    X ←top (STACK)
    while (not empty (STACK))
      begin
        if (X ∈ Σ and X = cc)
          cc ←PROSS
          pop (STACK)
        else if (X = S)
          if (cc = 'a' or cc = 'b' or cc = 'c')
            pop (STACK)
            push (PQ, STACK)
            output (S →PQ)
          else ERRORE (…)
        else if (X = P)
          if (cc = 'a')
            cc ←PROSS
            pop (STACK)
            push (Pb, STACK)
            output (P →aPb)
          else if (cc = 'b')
            cc ←PROSS
            pop (STACK)
            push (Pa, STACK)
            output (P →bPa)
          else if (cc = 'c')
            cc ←PROSS
            pop (STACK)
            output (P →c)
          else ERRORE (…)
```

```
Program esempio
  begin
    cc ←PROSS
    X ←top (STACK)
    while (not empty (STACK))
      begin
        if (X ∈ Σ and X = cc)
          cc ←PROSS
          pop (STACK)
        else if (X = variabile della grammatica)
          if (cc = 'a' or cc = '…' …) //insieme guida
            • Se Gui(X) = x & la parte dx della produz. inizia con x
                ▪ cc ← PROSS
                ▪ pop
                ▪ push(parte dx)
                • P → aBc
                • P → aBC
            • Se Gui(X) = x & la parte dx della produz. non inizia con x
                ▪ pop
                ▪ push(parte dx meno il primo terminale)
                • P → BC
                •
            • Se X = cc
                ▪ cc ← PROSS
                ▪ pop
                • P → c
            • Se P → ε
                ▪ pop
          else if (cc = '…' or cc = '…' …) //altro insieme guida
          …
          else ERRORE(..)
```

```
        else if (X = Q)
          if (cc = '&')
            cc ←PROSS
            pop (STACK)
            push (PQ, STACK)
            output (Q →&PQ)
          else if (cc = '$')
            pop (STACK)
            output (Q → ε)
          else ERRORE (…)
        else ERRORE (…)
        X ←top (STACK)
      end
    if (cc = '$')
      output ("stringa accettata")
    else ERRORE (…)
end
```

```
            else if (X = altra variabile della grammatica)
              …
            else ERRORE(..)
            X ←top (STACK)
        end
      if(cc = '$')
        output ("stringa accettata")
      else ERRORE(..)
    end
```

Analizzatore a discesa ricorsiva

Ad ogni variabile A con produzioni $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ si associa una procedura:

$$
\begin{array}{l}
\underline{Procedure\ A} \\
\quad \underline{begin} \\
\quad\quad \underline{if}(cc \in Gui(A \to \alpha_1)) \\
\quad\quad\quad body(\alpha_1) \\
\quad\quad \underline{else\ if}(cc \in Gui(A \to \alpha_2)) \\
\quad\quad\quad body(\alpha_2) \\
\quad\quad\quad \ldots \\
\quad\quad \underline{else\ if}(cc \in Gui(A \to \alpha_k)) \\
\quad\quad\quad body(\alpha_k) \\
\quad\quad \underline{else}\ ERRORE(..) \\
\quad \underline{end}
\end{array}
$$

- Se $\alpha = \varepsilon \to body(\varepsilon) = \underline{do\ nothing}$
- Se $\alpha = X_1 \ldots X_m \to body(X_1 \ldots X_m) = act(X_1)\ act(X_2) \ldots act(X_m)$

$$
act(X) = \begin{cases} \underline{call}(X) & se\ X\ \in V\ (\text{è una variabile}) \\ \underline{if}(cc = X) & \\ \quad cc \leftarrow PROSS & altrimenti\ (es. P \to a) \\ \underline{else}\ ERRORE(..) & \end{cases}
$$

$$
\begin{array}{l}
\underline{Program}\ discesa\_ricorsiva \\
\quad \underline{begin} \\
\quad\quad cc \leftarrow PROSS \\
\quad\quad \underline{call}(S) \\
\quad\quad \underline{if}(cc\ =\ \text{'\$'}) \\
\quad\quad\quad \text{"}stringa\ accettata\text{"} \\
\quad\quad \underline{else}\ ERRORE(\ldots) \\
\quad \underline{end}
\end{array}
$$

Scrivi sempre:

$$
\begin{array}{l}
\underline{Program}\ discesa\_ricorsiva \\
\quad \underline{begin} \\
\quad\quad cc \leftarrow PROSS \\
\quad\quad \underline{call}(start\ symbol) \\
\quad\quad \underline{if}(cc\ =\ \text{'\$'}) \\
\quad\quad\quad \text{"}stringa\ accettata\text{"} \\
\quad\quad \underline{else}\ ERRORE(\ldots) \\
\quad \underline{end}
\end{array}
$$

Scrivi una procedura per ogni variabile (compreso lo start symbol):

$\underline{Procedure}$ variabile

$\quad \underline{begin}$

$\quad\quad \underline{if}(cc = 'a' or\ cc = '..')$ //insieme guida della gramm.

- **Se c'è terminale** $\Rightarrow \underline{\boldsymbol{if}}(\boldsymbol{cc = terminale})$
  $\quad \boldsymbol{cc \leftarrow PROSS}$
  $\quad \ldots$
  $\underline{\boldsymbol{else}}\ \boldsymbol{ERRORE}$
- **Se c'è variabile** $\Rightarrow \underline{\boldsymbol{call}}(\boldsymbol{variabile})$
  - $\boldsymbol{Se\ A \to a \Longrightarrow cc \leftarrow PROSS}$ (non c'è if perché è quello dell'insieme Gui)
  - $\boldsymbol{Se\ A \to aBC \Longrightarrow cc \leftarrow PROSS,\ \underline{call}(B), \underline{call}(C)}$
  - $\boldsymbol{Se\ A \to aBc \Longrightarrow cc \leftarrow PROSS,\ \underline{call}(B)}$
    $\quad \boldsymbol{\underline{if}(cc = 'c')\ cc \leftarrow PROSS}$
    $\quad \boldsymbol{\underline{else}\ ERRORE(..)}$
  - $\boldsymbol{Se\ A \to BC \Longrightarrow \underline{call}(B), \underline{call}(C)}$
  - $\boldsymbol{Se\ A \to \varepsilon \Longrightarrow \underline{do\ nothing}}$

$\quad\quad \underline{else\ if}(cc = '..'\ or\ cc = '..')$ // altro insieme guida

$\quad\quad \ldots$

$\quad\quad \underline{else}\ ERRORE(..)$

$\quad \underline{end}$

$$
\begin{array}{ll}
S \to PQ & Gui(S \to PQ) = \{a, b, c\} \\
Q \to \&PQ & Gui(Q \to \&PQ) = \{\&\} \\
Q \to \varepsilon & Gui(Q \to \varepsilon) = \{\$\} \\
P \to aPb & Gui(P \to aPb) = \{a\} \\
P \to bPa & Gui(P \to bPa) = \{b\} \\
P \to c & Gui(P \to c) = \{c\}
\end{array}
$$

```
Program discesa_ricorsiva
  begin
    cc ← PROSS
    call(S)
    if(cc = '$')
      "stringa accettata"
    else ERRORE(...)
  end
```

```
Procedure S
  begin
    if(cc ∈ {a, b, c})
      call(P)
      call(Q)
    else ERRORE(...)
  end
```

```
Procedure Q
  begin
    if(cc = '&')
      cc ← PROSS
      call(P)
      call(Q)
    else if(cc = '$')
      do nothing
    else ERRORE(...)
  end
```

```
Procedure P
  begin
    if(cc = 'a')
      call(P)
      if(cc = 'b')
        cc ← PROSS
      else ERRORE(...)
    else if(cc = 'b')
      call(P)
      if(cc = 'a')
        cc ← PROSS
      else ERRORE(...)
    else if(cc = 'c')
      cc ← PROSS
    else ERRORE(...)
  end
```

Grammatiche LL(1): proprietà

1) **Senza ricorsioni sinistre**

   Una grammatica ricorsiva sx, cioè tale che per qualche non terminale A si ha una derivazione $A \Longrightarrow^{+} A\alpha$, non è $LL(1)$

   $$S \to Sa \mid b \quad \rightarrow \quad Gui(S \to Sa) = Gui(S \to b) = \{b\}$$

2) **Non ambigua**
3) Per ogni coppia di produzioni del tipo: $A \to \alpha$, $A \to \beta$
   - da α e da β non derivano stringhe che iniziano con lo stesso terminale
   - al più una tra α e β è annullabile
   - se α è annullabile, da β non deriva nessuna stringa che inizia con un terminale nell'insieme $FW(A)$

Grammatiche non LL(1)

Data una grammatica non $LL(1)$, è qualche volta possibile ottenerne una equivalente $LL(1)$.

In particolare si può:

1. **Eliminare le ricorsioni sinistre**, sia immediate sia non immediate, sostituendole con ricorsioni destre.

Eliminazione delle ricorsioni sinistre immediate				
$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_k$ $A \to \beta_1 \mid \beta_2 \mid \dots \mid \beta_h$	$k \geq 1, \alpha_i \neq \varepsilon$ $h \geq 1$	$\rightarrow$ $\rightarrow$	Con ε-produzioni $A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A'$ $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_k A' \mid \varepsilon$ Senza ε-produzioni $A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_h$ $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_k A' \mid \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$	$\boldsymbol{A \to A\alpha \mid \beta \;\Rightarrow\; A \to \beta A' \quad A' \to \alpha A' \mid \varepsilon}$

$$
\begin{array}{lcl}
E \to E + T \mid T & & E \to TE' \\
 & & E' \to +TE' \mid \varepsilon \\
T \to T * F \mid F & \rightarrow & T \to FT' \\
 & & T' \to *FT' \mid \varepsilon \\
F \to (E) \mid id & & F \to (E) \mid id
\end{array}
$$

2. **Fattorizzazione sinistra** cioè cercare di rendere la scelta della produzione da usare ad ogni passo deterministica posticipando, quando possibile, la scelta tra diverse alternative di riscrittura che hanno un prefisso comune.

Per ogni non terminale A si trova il massimo prefisso α comune a due o più alternative. Si sostituiscono tutte le produzioni $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_m$ con $A \to \alpha A'$, $A' \to \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$, lasciando le altre produzioni da A inalterate.

$A \to bD|eE$
$C \to cC|\varepsilon$
$D \to dE|\varepsilon$
$E \to Ca|a$

$a \in Gui(E \to Ca) \cap Gui(E \to a)$.
Sostituiamo a C le sue produzioni
→

$A \to bD|eE$
$C \to cC|\varepsilon$
$D \to dE|\varepsilon$
$E \to cCa|a$

La grammatica ottenuta è LL(1).

$A \to Ba|b$
$B \to bdC|bd|eC|e$
$C \to cC|\varepsilon$

$Gui(A \to Ba) = \{b, e\}$
$Gui(A \to b) = \{b\}$
$Gui(B \to bdC) = \{b\}$
$Gui(B \to bd) = \{b\}$
$Gui(B \to eC) = \{e\}$
$Gui(B \to e) = \{e\}$

Per ottenere una grammatica $LL(1)$ non basta la fattorizzazione sinistra sulle produzioni da B in quanto continuerebbe ad essere presente b negli insiemi guida di entrambe le produzioni da A. Eliminiamo la variabile B…

$A \to bdCa|bda|eCa|ea|b$
$C \to cC|\varepsilon$

12. Analisi ascendente.
Parsificazione LR: Simple LR

Stile generale di parsificazione bottom-up: *parsificazione shift-reduce*.

Avendo una grammatica G e una parola w, il parsificatore bottom-up cerca di "ridurre" la stringa w allo start symbol di G: ad ogni passo di riduzione una sottostringa che corrisponde al corpo di una produzione è sostituita dalla variabile di testa della produzione stessa.

Quando ridurre e quale produzione applicare?

Handle

E' una sottostringa che costituisce il corpo di una produzione, la cui riduzione rappresenta un passo della derivazione rightmost al contrario.

Definizione

Se $S \Rightarrow_R^* \alpha A w \Rightarrow_R \alpha\beta w$, la produzione $A \to \beta$ (o semplicemente β) nella posizione "dopo α" è un handle di $\alpha\beta w$.

Grammatica	Forma sentenziale	Handle	Produzione
$E \to E + T$	1. id*id	id	$F \to id$
$E \to T$	2. F*id	F	$T \to F$
$T \to T * F$	3. T*id	id	$F \to id$
$T \to F$	4. T*F	T*F	$T \to T * F$
$F \to (E)$	5. T	T	$E \to T$
$F \to id$	6. E		

La parsificazione shift-reduce è una forma di parsificazione bottom-up in cui uno stack contiene "i simboli della grammatica" e un buffer contiene la parte ancora da esaminare della stringa in input w: un automa push-down.

Il handle formato dai simboli sul top dello stack viene riconosciuto e sostituito con la variabile.

Configurazione iniziale			Configurazione finale	
Stack	**Input**	→	**Stack**	**Input**
$\$$	$w\$$		$\$S$	$\$$

Grammatica	Stack	Input	Azione
	$\$$	$id * id\$$	Azione
	$\$id$	$* id\$$	Riduci con $F \to id$
$E \to E + T$	$\$F$	$* id\$$	Riduci con $T \to F$
$E \to T$	$\$T$	$* id\$$	Sposta
$T \to T * F$	$\$T *$	$id\$$	Sposta
$T \to F$	$\$T * id$	$\$$	Riduci con $F \to id$
$F \to (E)$	$\$T * F$	$\$$	Riduci con $T \to T * F$
$F \to id$	$\$T$	$\$$	Riduci con $E \to T$
	$\$E$	$\$$	Accetta

Prefisso ammissibile

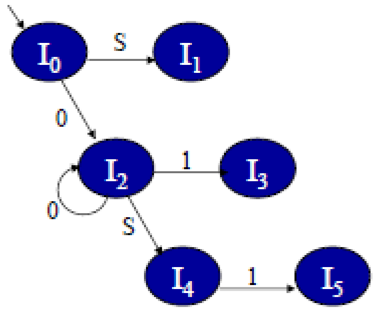
Come troviamo i membri destri delle produzioni da ridurre (gli handle)?

Ci basiamo sulla nozione di prefisso ammissibile.

Se $S \Rightarrow_R^* \alpha A w \Rightarrow_R \alpha\beta w$, è una derivazione rightmost e $\beta = \beta_1\beta_2$:

- $\alpha\beta_1$ è un prefisso ammissibile
- Se $\beta_2 = \varepsilon$, il prefisso è completo, ossia si è trovato un handle.

Il linguaggio dei prefissi ammissibili è un linguaggio regolare quindi si può costruire un automa a stati finiti: un automa che riconosce i prefissi ammissibili "memorizzando" nei suoi stati delle "stringhe" che ricordano quanta parte del membro destro di ogni produzione è stata letta.



Questo automa, i cui stati sono tutti finali, riconosce i prefissi 0, 00, 00…0, 01, 001, 00…01, 0S, 0S1, 00S1,...cioè i prefissi ammissibili.

A partire da esso si può costruire l'automa shift-reduce che ricorda nello stack i simboli della grammatica (o meglio gli stati dell'automa finito rappresentativi di tali simboli), ed esegue le riduzioni.

Ai due stati I_3 e I_5 vengono associate le riduzioni derivanti dalle produzioni $S \to 01$ e $S \to 0S1$, rispettivamente, mentre agli altri stati sono associati gli spostamenti (shift) della testina sul nastro di input.

N.B. le transizioni entranti in uno stato hanno tutte la stessa etichetta, pertanto lo stato è rappresentativo anche del simbolo della grammatica che etichetta tali transizioni.

La figura seguente dà un'idea di come l'automa riconoscitore dei prefissi ammissibili e lo stack possano essere usati per prendere le decisioni shift- reduce dell'algoritmo di parsificazione.

Linea	Stack	Simboli	Input	Azione
1	I_0	$\$$	$000111\$$	$Shift$ (sposta la testina sul nastro di input)
2	$I_0 I_2$	$\$0$	$00111\$$	$Shift$
3	$I_0 I_2 I_2$	$\$00$	$0111\$$	$Shift$
4	$I_0 I_2 I_2 I_2$	$\$000$	$111\$$	$Shift$
5	$I_0 I_2 I_2 I_2 I_3$	$\$0001$	$11\$$	$Riduci\ con\ \ S \to 01;\ pop;\ pop;\ push\ L_4(S)$
6	$I_0 I_2 I_2 I_4$	$\$00S$	$11\$$	$Shift$
7	$I_0 I_2 I_2 I_4 I_5$	$\$00S1$	$1\$$	$Riduci\ con\ \ S \to 0S1;\ pop;\ pop;\ pop;\ push\ L_4(S)$
8	$I_0 I_2 I_4$	$\$0S$	$1\$$	$Shift$
9	$I_0 I_2 I_4 I_5$	$\$0S1$	$\$$	$Riduci\ con\ \ S \to 0S1;\ pop;\ pop;\ pop;\ push\ L_1(S)$
10	$I_0 I_1$	$\$S$	$\$$	Accetta

Costruzione dell'automa riconoscitore dei prefissi ammissibili (automa LR(0))

Gli stati tengono traccia di dove si è nel processo di parsificazione e sono formati da insiemi di "item".
Un **item** per una grammatica G è una produzione di G con un punto in qualche posizione del suo membro destro.
Un item indica quanta parte di una produzione è stata presa in esame in un certo momento del processo di parsificazione.

Per esempio $S \to \cdot\, 0S1$ indica che ci si aspetta di trovare sul nastro di input una stringa derivabile da $0S1$, $S \to 0 \cdot S1$ indica invece che 0 è già stato letto e che ci si aspetta ora di trovare una stringa derivabile da $S1$.

Dato un insieme di item I, definiamo "**chiusura di I**" l'insieme costruito a partire da I aggiungendo per ogni item della forma $A \to \alpha \cdot B\beta$ gli item $B \to \cdot\, \gamma_1, B \to \cdot\, \gamma_2, \dots, B \to \cdot\, \gamma_k$ se le produzioni da B sono $B \to \gamma_1, B \to \gamma_2, \dots, B \to \gamma_k$, fino a quando l'insieme di item resta inalterato.

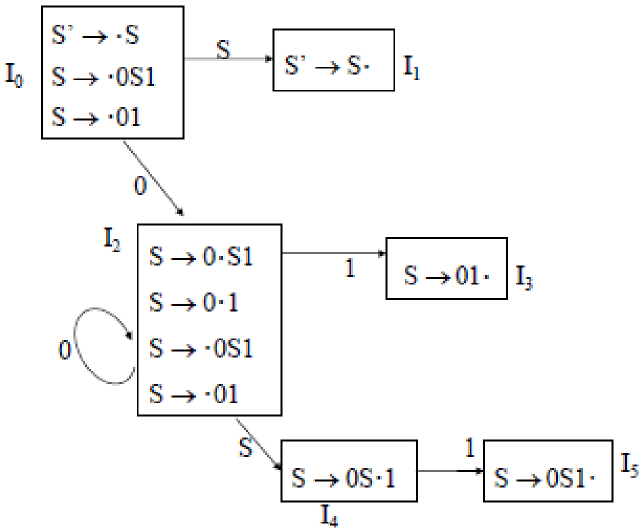
La produzione $A \to \varepsilon$ produce il solo item $A \to \cdot$.
Nella tabella di parsificazione non si mette la colonna ε nelle azioni.

N.B. Lo scopo della nuova produzione è di indicare al parsificatore il punto in cui si deve fermare e accettare l'input; l'accettazione di una stringa avviene quando e solo quando il parsificatore riduce per mezzo di $S' \to S$.

Uno stato è chiamato:
- di spostamento se contiene solo item della forma $A \to \beta_1 \cdot \beta_2\ (\beta_2 \neq \varepsilon)$
- di riduzione se contiene un unico item della forma $A \to \beta\ \cdot$
- inadeguato se contiene item di entrambe le forme

Una grammatica è SLR(0) (Simple LR(0)) **se nessuno stato del riconoscitore dei prefissi ascendenti è inadeguato.**
Un linguaggio non prefix-free non è SLR(0).

Algoritmo shift/reduce



Vediamo come l'automa $LR(0)$ fornisce supporto per le decisioni shift/reduce dell'automa a pila, riconoscitore del linguaggio. Se l'automa $LR(0)$ è in uno stato j che ha una transizione etichettata con un simbolo di input a, si deve mettere a (o meglio lo stato che ricorda a) sulla pila dell'automa push-down, altrimenti si deve effettuare una riduzione e gli item nello stato j indicheranno quale produzione usare.

Se in uno stato ci sono item sia che suggeriscono una riduzione sia che suggeriscono uno spostamento, è possibile capire cosa fare? Come scegliere?
Esaminiamo le grammatiche nelle quali è possibile capire se fare uno spostamento o una riduzione "guardando avanti" un simbolo: la riduzione deve essere fatta se il carattere in input è uno di quelli che può seguire la variabile di testa, A, dell'item $A \to \alpha \cdot$, cioè appartiene all'insieme $FOLLOW(A)$, altrimenti si deve avanzare sul nastro di input.

<u>Gli stati dell'automa LR(0) sono tutti finali.</u>

Una grammatica il cui automa $LR(0)$ non presenta conflitti shift/reduce e reduce/reduce si chiama $SLR(1)$.

Vi sono grammatiche libere dal contesto a cui la parsificazione $SLR(1)$ non può essere applicata: noto il contenuto dello stack e il prossimo simbolo in input non si e` in grado di decidere se aggiungere il simbolo (lo stato) sullo stack o applicare una riduzione (conflitto shift/ reduce) oppure non si è in grado di decidere quale tra più possibili riduzioni effettuare (conflitto reduce/reduce).

Per una grammatica SLR(1) si può costruire un automa push-down deterministico definito sostanzialmente in questo modo:

$$< \{q\}, \Sigma, Q_F, d, q, I_0 >$$

- $\delta(q, a, I_k) = \{(q, I_k I_j)\}$ se $A \to \alpha \cdot a\beta$ è in I_k, $a \in \Sigma$ e $\delta_F(I_k, a) = I_j$
- Se $A \to \beta \cdot$ è in I_k e $\beta = x_1 \dots x_n$ $(|\beta| = n)$, lo stack deve contenere come ultimi n elementi $I_{k-n+1}, I_{k-n+1}, \dots, I_k$, rappresentativi dei simboli $x_1, x_2, \dots, x_n$. Se il simbolo in input a è in $FW(A)$, si cancellano dalla pila gli n elementi e si impila I_j se $\delta_F(I_{k-n}, A) = I_j$.
- Se $S' \to S \cdot$ è in I_k, la pila contiene $I_0 I_k$ è il simbolo in input è \$, "accetta".

E' utile rappresentare la funzione di transizione δ con una tabella di parsificazione in cui alle colonne vengono associati i simboli di $\Sigma \cup V$ e alle righe gli stati dell'automa $LR(0)$.
Numeriamo le produzioni di G da 1 a n.
Suddividiamo la tabella in due parti, $azioni$ e $goto$.
La parte azioni contiene le colonne associate ai simboli terminali e contiene le azioni $shift$ o $reduce$, la parte $goto$ individua le colonne associate ai simboli non terminali e contiene l'indicazione dello stato da aggiungere sul top dello stack, ossia lo stato rappresentativo della testa della produzione il cui corpo è stato ridotto.

- Se $A \to \alpha \cdot a\beta$ è in $I_k, \delta_F(I_k, a) = I_j$ e a è un simbolo terminale $\Longrightarrow \underline{azione}[k, a] = sj \quad (shift\ I_j)$
- Se $A \to \alpha \cdot$ è in I_k, per tutti gli a in FW(A) $\Longrightarrow \underline{azione}[k, a] = rn$
  $(reduce\ A \to \alpha$, se n è il numro associato alla produzione $A \to \alpha)$
- Se $S' \to S \cdot$ è in $I_k \Longrightarrow \underline{azione}[k, \$] = "accetta"$
- Se $\delta_F(I_k, a) = I_j$, A non terminale, $\underline{goto}[k, A] = j$

Tabella di parsificazione

- s_i significa impila lo stato *i*
- r_j significa riduci usando la produzione numero *i*
- le caselle vuote significano un errore

- Gli r_j vanno nelle colonne corrispondenti ai terminali $\in FW(X)$ dove X è l'elemento a sx della produzione.

Grammatica

$E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

Follow

$FW(E) = \{+,), \$\}$
$FW(T) = \{*, +,), \$\}$
$FW(F) = \{*, +,), \$\}$

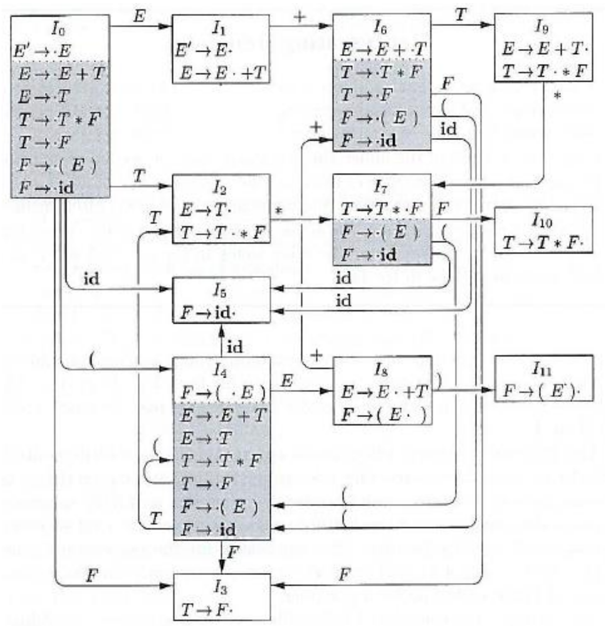


Tabella di parsificazione

Stati	Azioni						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accetta			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Comportamento dell'automa push-down sulla stringa: id*id+id$

	Stack	Input	Azione
1	0	id*id+id$	s5
2	0 5	*id+id$	r6
3	0 3	*id+id$	r4
4	0 2	*id+id$	s7
5	0 2 7	id+id$	s5
6	0 2 7 5	+id$	r6
7	0 2 7 10	+id$	r3
8	0 2	+id$	r2
9	0 1	+id$	s6
10	0 1 6	id$	s5
11	0 1 6 5	$	r6
12	0 1 6 3	$	r4
13	0 1 6 9	$	1
14	0 1	$	accetta

Parti da Stack 0 e in input la stringa. Guarda cosa c'è nella cella [stato,AZ]. Se c'è:

- *si*:
  - aggiungi sullo stack I_i
  - shifti sulla stringa
- *ri*:
  - togli tanti simboli sullo stack quant'è la lunghezza della parte dx della produzione #i
  - guardi ora nella cella [$stack_{nuovo}$, GO(testa di *i*)] e metti sul top quello stato

Se nella tabella così costruita si presentano più azioni in qualche posizione, la grammatica non è $SLR(1)$.

Algoritmo di parsificazione SLR(1)

```
Program SLR
   begin
   cc ←PROSS //primo carattere di w
      while(true)
         k ←top (STACK)
         if(AZ[k, cc] = shift i)
            push(I_i , STACK)
            cc ←PROSS
         else if(AZ[k, cc] = r n) //n è il numero delle produzioni
            if(n = '1') //produzione #1: S → b (lo fai ∀produz. della gramm.)
               pop(STACK) |b| volte //b = lunghezza della parte dx della produz. #1
                  if(top(STACK) ∈ {j, …}) //j è il numero dello stato I che trovi guardando nella tabella di parsific. nelle righe
                     push(GO[k, S], STACK)                in cui la colonna S (parte sx della prod. #1) ha dei valori
                  else ERRORE
            if(n = '…')
               ….
         else if(AZ[k, cc] = 'accetta') //parsificazione finita, stringa accettata
            break
         else ERRORE
end
```

13. Traduzione guidata dalla sintassi

Definizione guidata dalla sintassi (SDD)

	SINTASSI		SEMANTICA
Grammatica	Simboli (terminali e non)	Si associano (+)	Attributi
	Produzioni (regole sintattiche)	Si associano (+)	Regole semantiche
	parole		significato

Notazione infissa → Notazione postfissa

Produzioni	Regole semantiche
$E \rightarrow E_1 + T$	E.*code* = E_1.*code* || T.*code* || '+'
$E \rightarrow E_1 - T$	E.*code* = E_1.*code* || T.*code* || '-'
$E \rightarrow T$	E.*code* = T.*code*
$T \rightarrow$ digit	T.*code* = digit.*lexval*

- l'indice 1 in E_1 distingue le diverse occorrenze di E nelle produzioni
- || indica la concatenazione tra stringhe
- .*lexval* indica il valore lessicale (valore che si trova nella symbol table)
- .*code* è una stringa e si usa per i non terminali
- **.*code* per E e T è sintetizzato.**
- **.*lexval* è un valore fornito dall'analizzatore lessicale**

Gli attributi possono essere di un tipo qualunque: numeri, tipi, riferimenti a tabelle, stringhe, sequenze di codice, …

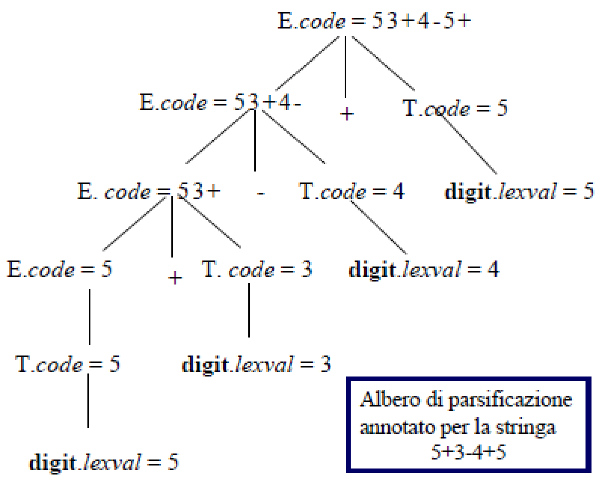
Se *X* è un simbolo e *a* un suo attributo, usiamo *X.a* per denotare il valore di *a* in un particolare nodo dell'albero di parsificazione etichettato *X*.

Gli attributi per i simboli terminali hanno i valori lessicali forniti dall'analizzatore lessicale. Nelle SDD non vi sono regole semantiche per calcolare i valori degli attributi per i terminali.

Per i simboli non terminali consideriamo due tipi di attributi:

- **Sintetizzati (↑):** un attributo sintetizzato per una variabile *A* in un nodo *n* dell'albero di parsificazione è definito da una regola semantica associata alla produzione in *n* e *il suo valore è calcolato solo in termini dei valori degli attributi nei nodi figli di n e in n stesso*.
  (*A* è il simbolo a sinistra nella produzione, cioè la testa).
- **Ereditati (↓):** un attributo ereditato per una variabile *A* in un nodo *n* dell'albero di parsificazione è definito da una regola semantica associata alla produzione nel nodo padre di *n* e *il suo valore è calcolato solo in termini dei valori degli attributi del padre di n, di n stesso e dei suoi fratelli*.
  (*A* è un simbolo nel corpo della produzione, cioè al membro destro).

Per visualizzare la traduzione specificata da un SDD può essere utile usare gli alberi di parsificazione, costruendo prima l'albero e poi usando le regole per valutare gli attributi in ogni nodo dell'albero.
Un albero di parsificazione che mostra i valori degli attributi è chiamato **albero di parsificazione annotato**.



Albero di parsificazione annotato per la stringa 5+3-4+5

Come costruire un albero annotato:

- prima di valutare un attributo in un nodo, si devono valutare gli attributi dai quali dipende il suo valore
- gli attributi sintetizzati possono essere valutati in ordine bottom-up
- negli SDD che hanno sia attributi sintetizzati sia ereditati, non vi è garanzia che vi sia almeno un ordine in cui valutare gli attributi nei nodi perché potrebbero essere presenti regole "circolari" che rendono impossibile la valutazione.

Il problema di determinare se, data una SDD, esiste una circolarità in qualche albero di parsificazione da annotare è decidibile, ma ha complessità esponenziale in tempo.

Dato però un albero di parsificazione si può costruire un **grafo delle dipendenze** che mostri il flusso di informazione tra le istanze degli attributi. Tale grafo permette di scoprire se la valutazione è possibile e in tal caso trovare un ordine per la valutazione stessa.

$.a \rightarrow .b \implies .b$ dipende da $.a$

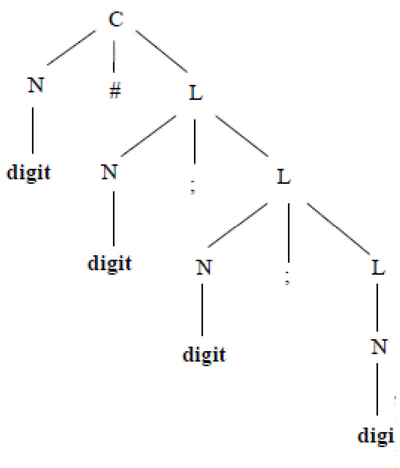
Se il grafo presenta un ciclo non è possibile valutare gli attributi.
L'ordine di valutazione deve rispettare un ordinamento topologico dei vertici del grafo.

La grammatica genera frasi formate da un numero seguito da #, a sua volta seguito da una sequenza di numeri separati da ";". La definizione syntax-directed associa ad ogni frase della grammatica la lista dei numeri ottenuti sottraendo il primo a tutti gli altri.

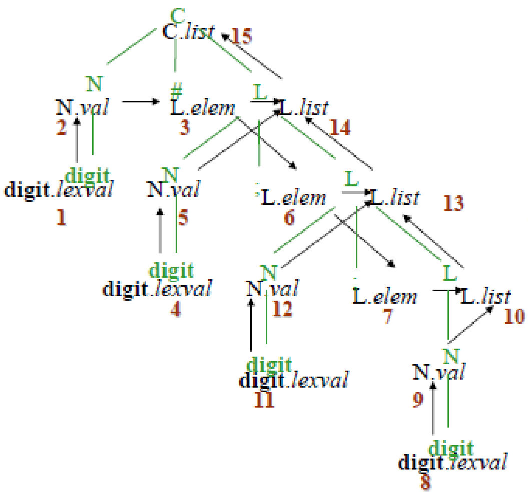
Produzioni	**Regole semantiche**
$C \rightarrow N\#L$	$C.list = L.list\,;\quad L.elem = N.val$
$L \rightarrow N; L_1$	$L.list = \underline{cons}\,(N.val - L.elem,\, L_1.list)\,;\quad L_1.elem = L.elem$
$L \rightarrow N$	$L.list = \underline{cons}\,(N.val - L.elem,\, \underline{null})$
$N \rightarrow \textbf{digit}$	$N.val = \textbf{digit}.lexval$

- $.list, .val$: sintetizzati
- $.elem$: ereditato
- $\underline{cons(a, lista)}$: funzione che inserisce a in testa alla $lista$

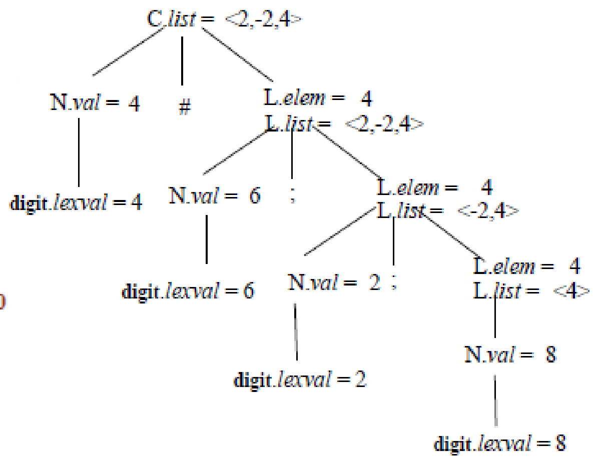
Albero di parsificazione per la stringa:
$digit\#digit; digit; digit$



Grafo delle dipendenze



Albero annotato per la parola $4\#6; 2; 8$ **fornita al parser dal lexer come** $digit\#digit; digit; digit$.
La traduzione di $4\#6; 2; 8$ **è la lista** $< 2, -2, 4 >$



14. Schemi di traduzione e Definizioni S(L)-attribuite

Schemi di traduzione diretti dalla sintassi (SDT)

Uno **schema di traduzione diretto dalla sintassi** è una definizione guidata dalla sintassi in cui le azioni semantiche, racchiuse tra parentesi graffe, sono inserite nei membri destri delle produzioni nella posizione in cui devono essere eseguite durante la *parsificazione* della grammatica.
Le azioni semantiche specificate dagli SDT possono essere rappresentate dalla valutazione degli attributi di una SDD, ma anche da *frammenti di programma* e in generale *operazioni con side-effects* (come le operazioni di output).
Oltre alla valutazione degli attributi , negli schemi di traduzione si possono inserire frammenti di codice o azioni di input-output.

Dalle SDD (Definizione guidata dalla sintassi) alle SDT (Schemi di traduzione diretti dalla sintassi)

Una SDT si può vedere come un'implementazione di una SDD.
Vi sono due classi di SDD che sono realizzabili mediante SDT:

- **SDD S-attribuita**
  - Presentano solo attributi sintetizzati (dipendono da figli e se stesso) → attributi a destra nelle produzioni
  - Sono implementabili da una SDT associata a un parser bottom-up (↑)
  - In un'analisi bottom-up le az. semantiche vengono inserite alla fine delle produz. e realizzate durante la loro riduzione
- **SDD L-attribuita**
  - Presentano sia attributi sintetizzati che attributi ereditati (dipendono da padre, fratelli e se stesso)
  - Gli attributi ereditati devono soddisfare il vincolo: **(gli attributi possono dipendere solo dai fratelli a sx o dal padre)**
    Per ogni produzione $A \to X_1X_2 \dots X_n$ ogni attributo ereditato di X_j dipende solo da:
    - attributi ereditati o sintetizzati dei simboli $X_1X_2 \dots X_{j-1}$ a sinistra di X_j nella produzione
    - attributi ereditati di A
    - attributi ereditati o sintetizzati di X_j, purchè non vi siano cicli nel grafo delle dipendenze formati dagli attributi di questa occorrenza di X_j

    Nel grafo delle dipendenze gli archi tra gli attributi associati a una produzione possono andare da sinistra a destra, ma non da destra a sinistra (left to right →, ↚).
  - Sono implementabili mediante SDT associate a un parser top-down (↓)

Queste classi garantiscono l'esistenza di un ordine di valutazione poiché non ammettono grafi di dipendenza ciclici e le regole possono essere convertite in un SDT con azioni specificate al momento giusto.

Nota: ogni SDD S-attribuita è anche L-attribuita ma non il contrario.

Esempio di SDD S-attribuita: calcolo del valore di un'espressione:

SDD:			⟹ SDT:
	$E \to E_1 + T$	$E.val = E_1.val + T.val$	$E \to E_1 + T \ \{E.val = E_1.val + T.val\}$
	$E \to E_1 - T$	$E.val = E_1.val - T.val$	$E \to E_1 - T \ \{E.val = E_1.val - T.val\}$
	$E \to T$	$E.val = T.val$	$E \to T \ \{E.val = T.val\}$
	$T \to digit$	$T.val = digit.lexval$	$T \to digit \ \{T.val = digit.lexval\}$

Nella SDT le azioni semantiche vengono inserite alla fine delle produzioni e realizzate durante la loro riduzione.

Esempio: traduzione da forma infissa a forma postfissa

SDD: (S-attribuita)			⟹ SDT:
	$E \to E_1 + T$	$E.code = E_1.code \mid\mid T.code \mid\mid '+'$	$E \to E_1 + T \ \{output('+')\}$
	$E \to E_1 - T$	$E.code = E_1.code \mid\mid T.code \mid\mid '-'$	$E \to E_1 - T \ \{output('-')\}$
	$E \to T$	$E.code = T.code$	$E \to T$
	$T \to digit$	$T.code = digit.lexval$	$T \to digit \ \{output(digit.lexval)\}$

Da SDD L-attribuita a SDT

(1) Inserire le azioni che calcolano gli attributi ereditati per un non terminale A immediatamente prima dell'occorrenza di A nel corpo della produzione.

(2) Se diversi attributi ereditati per A dipendono uno dall'altro, ordinare la valutazione degli attributi in modo che quelli necessari prima siano calcolati per primi.

(3) Porre la azioni che calcolano un attributo sintetizzato per la variabile a sinistra in una produzione alla fine del corpo della produzione stessa.

$C \rightarrow N\#\ \{L.elem = N.val\}\ L\ \{C.list = L.list\}$
$L \rightarrow N;\ \{L_1.elem = L.elem\}\ L_1\ \{L.list = \underline{cons}\,(N.val - L.elem,\ L_1.list)\}$
$L \rightarrow \varepsilon\ \{L.list = \underline{null}\}$
$N \rightarrow digit\ \{N.val = digit.lexval\}$

La grammatica è LL(1):	**Insiemi guida**
$C \rightarrow N\#L$	$\{digit\}$
$L \rightarrow N;L_1$	$\{digit\}$
$L \rightarrow \varepsilon$	$\{\dashv\}$
$N \rightarrow digit$	$\{digit\}$

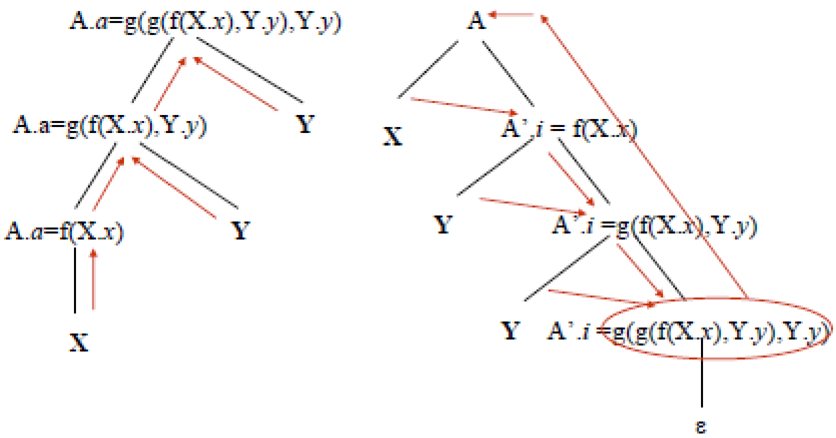
La grammatica è stata ottenuta da quella di un esempio visto in precedenza sostituendo la produzione $L \rightarrow N$ con $L \rightarrow \varepsilon$ e modificando di conseguenza la regola semantica associata. Questa grammatica, a differenza della precedente, è LL(1). La funzione $\underline{cons}$ aggiunge un elemento in testa a una lista.

Nessuna grammatica con ricorsione sinistra può essere parsificata in modo top-down deterministicamente.

Se l'SDD è S-attribuita, eliminando la ricorsione sinistra si può sempre costruire un SDT mettendo le azioni per il calcolo degli attributi nelle appropriate posizioni all'interno delle nuove produzioni.

$A \rightarrow A_1Y\ \ \{A.a = g(A_1.a,\ Y.y)\}$
$A \rightarrow X\ \ \{A.a = f(X.x)\}$

$A \rightarrow X\ \{A'.i = f(X.x)\}\ A'\ \{A.a = A'.s\}$
$A' \rightarrow Y\ \{A'_1.i = g(A'.i,\ Y.y)\}\ A'_1\ \{A'.s = A'_1.s\}$
$A' \rightarrow \varepsilon\ \{A'.s = A'.i\}$



Espressioni aritmetiche

$E \rightarrow E_1 + T$	$\{E.val = E_1.val + T.val\}$
$E \rightarrow E_1 - T$	$\{E.val = E_1.val - T.val\}$
$E \rightarrow T$	$\{E.val = T.val\}$
$T \rightarrow (\ E\ )$	$\{T.val = E.val\}$
$T \rightarrow \mathbf{num}$	$\{T.val = \mathbf{num}.val\}$

questa grammatica, ricorsiva a sinistra, non è LL(1)

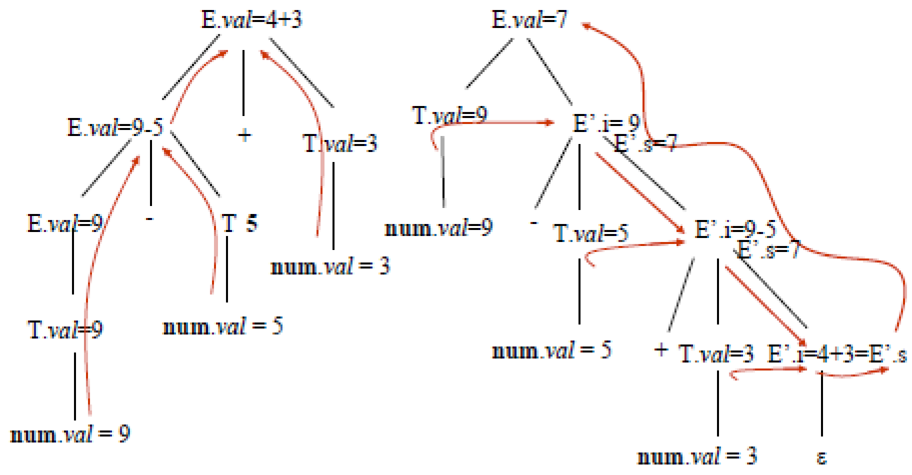
$E \rightarrow TE'$
$E' \rightarrow +TE'_1$
$E' \rightarrow -TE'_1$
$E' \rightarrow \varepsilon$
$T \rightarrow (E)$
$T \rightarrow \mathbf{num}$

questa grammatica, ricorsiva a destra è LL(1)
Come definire regole semantiche per calcolare E.*val*?

Espressioni aritmetiche

$E \rightarrow T\ \{E.val = T.val\}$	$E \rightarrow T$ $\quad E'$	$\{E'.i = T.val\}$ $\{E.val = E'.s\}$
$E \rightarrow E_1 + T\ \{E.val = E_1.val + T.val\}$	$E' \rightarrow +$ $\quad T$ $\quad E'_1$	 $\{E'_1.i = E'.i + T.val\}$ $\{E'.s = E'_1.s\}$
$E \rightarrow E_1 - T\ \{E.val = E_1.val - T.val\}$	$E' \rightarrow -$ $\quad T$ $\quad E'_1$	 $\{E'_1.i = E'.i - T.val\}$ $\{E'.s = E'_1.s\}$
	$E' \rightarrow \varepsilon$	$\{E'.s = E'.i\}$
$T \rightarrow (\ E\ )\ \{T.val = E.val\}$	$T \rightarrow (E)$	$\{T.val = E.val\}$
$T \rightarrow \mathbf{num}\ \{T.val = \mathbf{num}.val\}$	$T \rightarrow \mathbf{num}$	$\{T.val = \mathbf{num}.val\}$

Espressioni aritmetiche



E'.*i* e E'.*s* sono gli attributi ereditato e sintetizzato rispettivamente che 'operano' come E.*val* nella grammatica con ricorsione sinistra.

Gli schemi di traduzione possono essere usati per implementare SDD durante la parsificazione in due casi importanti:

- SDD S-attribuita e grammatica LR-parsificabile
- SDD L-attribuita e grammatica LL-parsificabile

15. Valutazione deterministica di definizioni L-attribuite e S-attribuite

Valutazione deterministica di definizioni L-attribuite

L'analizzatore a discesa ricorsiva per grammatiche LL(1) può essere modificato in modo da valutare gli L-attributi. Durante la parsificazione un'azione semantica nel corpo di una produzione (nello schema di traduzione) è eseguita non appena tutti i simboli della grammatica a sinistra dell'azione sono stati (presi in esame).

Ad ogni non terminale si associa una funzione che ha come parametri i valori degli attributi ereditati della variabile e restituisce i valori dei suoi attributi sintetizzati.

La funzione per un non terminale ha una variabile locale per ogni attributo ereditato o sintetizzato per i simboli che compaiono nelle parti destre delle produzioni da quel non terminale.

```
Program traduzione_discesa_ricorsiva
    begin
        cc ⟵ PROSS
        risultato ⟵ S
        if(cc = '$')
            print("Stringa corretta, la sua traduzione è: "risultato"")
        else ERRORE(…)
    end
```

```
function A(e1, . . . en)
    var s1, . . . , sm, X1_x1, . . . , X1_xk, . . . , Xh_x1, . . . , Xh_xr
    begin
        if(cc ∈ Gui(A → a1))
            body'(a1)
        else if(cc ∈ Gui(A → a2))
            body'(a2)
        ….
        else if(cc ∈ Gui(A → ak))
            body'(ak)
        else ERRORE(…)
        return < s1, . . . , sm>
end
```

Codice per le parti destre delle produzioni (body'(a_i)):

- Per ogni non terminale B si genera un'assegnazione *c ⟵ B(b_1, . . . b_n)*, che è una chiamata alla funzione associata a B
- Per ogni terminale i valori degli attributi vengono assegnati alle corrispondenti variabili e l'esame passa al simbolo dopo
- Le azioni semantiche vengono ricopiate dopo aver sostituito i riferimenti agli attributi con le variabili corrispondenti

Nel seguente schema di traduzione (SDT) < , > è una coppia e p_1, p_2 sono le proiezioni cioè p_1(D.*val*) e p_2(D.*val*) individuano la prima e la seconda componente dell'attributo D.*val*, rispettivamente.

$$\begin{aligned}
N &\to D\ K\ \{N.val = < p_1(D.val)\ x\ 2^{\boldsymbol{p_2(K.val)}} + p_1(K.val), p_2(K.val) >\} \\
K &\to N\ \{K.val = < p_1(N.val), p_2(N.val) + 1 >\} \\
K &\to \varepsilon\ \{K.val = < 0, 0 >\} \\
D &\to 0\ \{D.val = < 0, 0 >\} \\
D &\to 1\ \{D.val = < 1, 0 >\}
\end{aligned}$$

$$\left.\begin{aligned}
Gui(N \to DK) &= \{0,1\} \\
Gui(K \to N) &= \{0,1\} \\
Gui(K \to \varepsilon) &= \{\$\} \\
Gui(D \to 0) &= \{0\} \\
Gui(D \to 1) &= \{1\}
\end{aligned}\right. \Rightarrow LL(1)$$

```
function N
    var val, K_val, D_val
    begin
        if(cc ∈ {0,1})
            D_val ← D
            K_val ← K
            val ← <p1(D_val) x 2^p2(K_val) + p1(K_val), p2(K_val)>
        else ERRORE (…)
        return val
    end
```

```
function D
    var val
    begin
        if(cc = 0)
            val ← < 0,0>
            cc ← PROSS
        else if(cc = 1)
            val ← <1,0>
            cc ← PROSS
        else ERRORE (…)
        return val
    end
```

```
function K
    var val, N_val
    begin
        if(cc ∈ {0,1})
            N_val ← N
            val ← <p1(N_val), p2(N_val) + 1>
        else if(cc = $ )
            val ← < 0,0>
        else ERRORE (…)
        return val
    end
```

Lista delle differenze	**Insiemi Guida** $\Rightarrow LL(1)$
$C \rightarrow N\#\{L.elem = N.val\}\ L\ \{C.list = L.list\}$	$\{digit\}$
$L \rightarrow N;\ \{L_1.elem = L.elem\}\ L_1\ \{L.list = \underline{cons}\,(N.val - L.elem,\ L_1.list)\}$	$\{digit\}$
$L \rightarrow \varepsilon\ \{L.list = \underline{null}\}$	$\{\$\}$
$N \rightarrow digit\ \{N.val = digit.lexval\}$	$\{digit\}$

- **.elem** = ereditato **(parametro)**
- **.val** = sintetizzato
- **.list** = sintetizzato

```
Program trad_discesa_ricors
  begin
    cc ← PROSS
    list ← C
    if(cc = '$')
       "stringa accettata"
    else ERRORE(…)
  end

function N
  var val
  begin
    if(cc = digit)
      val ← digit.val
      cc ← PROSS
    else ERRORE(…)
    return val
  end
```

```
function L(elem)
  var list, N_val, L_elem, L_list
  begin
    if(cc = digit)
      N_val ← N
      if(cc = ';')
        cc ← PROSS
        L_elem ← elem
        L_list ← L(L_elem)
        list ← cons (N_val – elem, L_list)
      else ERRORE(…)
    else if(cc = '$')
      list ← null
    else ERRORE(…)
    return list
  end
```

```
function C
  var list, N_val, L_elem, L_list
  begin
    if(cc = digit)
      N_val ← N
      if(cc = '#')
        cc ← PROSS
        L_elem ← N_val
        L_list ← L(L_elem)
        list ← L_list
      else ERRORE(…)
    else ERRORE(…)
    return list
  end
```

- **Ogni volta che c'è una variabile $X \implies X\_val \leftarrow X$**
- **Ogni volta che c'è un terminale $a \implies if(cc = a)\ \ cc \leftarrow PROSS$**
- **Ogni volta che c'è una regola semantica $\{\ X.val = \cdots\} \implies X\_val = \cdots$**
- **Gli argomenti di $A()$ sono gli attributi ereditati di A**
- **Il valore restituito da $A()$ è l'insieme degli attributi sintetizzati di A**
- **C'è una variabile locale per ogni attributo ereditato o sintetizzato per i simboli che compaiono nelle parti dx delle produzioni di A**
- **Appena incontro una {} → scrivo il contenuto**
- **Se la parentesi era di un attributo ereditato (cioè si trova prima di un altro pezzo di produzione dx) → associo ad una variabile sintetizzata il valore restituito dalla funzione con parametro l'attributo ereditato (es: $F\_s = F(f\_e)$)**
- **Scrivo il contenuto della parentesi successiva {}**

- Per ogni non terminale B si genera un'assegnazione $C \leftarrow B(b_1, .., b_n)$ dove $b_1, .., b_n$ sono attributi ereditati di B.
- Per ogni terminale i valori degli attributi vengono assegnati alle corrispondenti variabili e l'esame passa la simbolo dopo.
- Le azioni semantiche vengono ricopiate dopo aver sostituito i riferimenti agli attributi con le variabili corrispondenti.

Valutazione bottom-up di SDD S-attribuite

Nelle SDD S-attribuite si hanno regole semantiche del tipo:

$$A \rightarrow X_1 \ldots X_n \qquad A.s = f(X_1.s, \ldots, X_n.s)$$

l'azione che calcola A.s è collocata al fondo, quando gli attributi di $X_1, \ldots, X_n$, sono stati calcolati, cioè quando viene effettuata la riduzione di $X_1 \ldots X_n$ ad A.

L'ordine di valutazione degli attributi corrisponde all'ordine delle riduzioni del parser bottom-up.

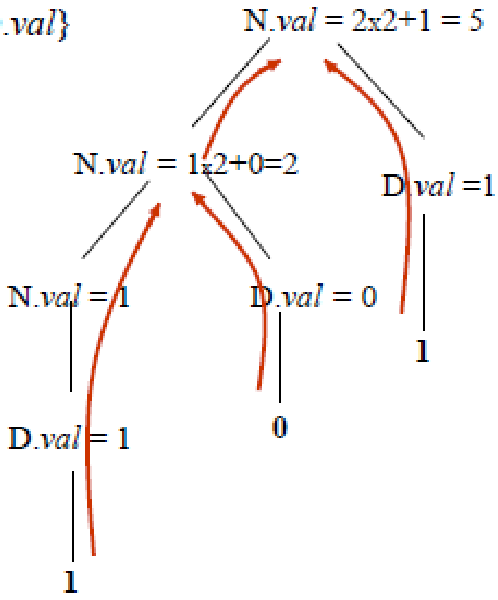
Gli attributi di un non terminale A possono essere valutati in corrispondenza di un passo di riduzione ad A.

$N \rightarrow N_1 D \qquad \{N.val = N_1.val \times 2 + D.val\}$

$N \rightarrow D \qquad \{N.val = D.val\}$

$D \rightarrow 0 \qquad \{D.val = 0\}$

$D \rightarrow 1 \qquad \{D.val = 1\}$



16. Generazione di codice intermedio

Nel processo di traduzione di un programma in un linguaggio sorgente in codice per una macchina target, il compilatore può costruire una sequenza di rappresentazioni intermedie.
Il linguaggio considerato è un frammento minimale di un linguaggio di programmazione imperativo con semplici espressioni aritmetiche e booleane e istruzioni di assegnazione, condizionali e iterazione.
Il linguaggio target è un codice postfisso rappresentato da un piccolo sottoinsieme del JAVA bytecode.

Espressioni aritmetiche

- `iload var` → carica sullo stack il valore intero *var*
- `istore var` → carica nella variabile *var* il valore intero che c'è sul top dello stack
- `iadd` → somma i due valori sul top dello stack e mette il risultato sul top dello stack
- `imul` → moltiplica i due valori sul top dello stack e mette il risultato sul top dello stack
- `ineg` → rende negativo il valore sul top dello stack e mette il risultato sul top dello stack
- `if_ cmpeq label` → se gli interi sul top dello stack sono **uguali** allora va all'istruzione *label* (salto condizionato)
- `if_cmpne label` → se gli interi sul top dello stack sono **diversi** allora va all'istruzione *label* (salto condizionato)
- `if_ cmple label` → se gli interi sul top dello stack sono **<=** allora va all'istruzione *label* (salto condizionato)
- `if_cmplt label` → se gli interi sul top dello stack sono **<** allora va all'istruzione *label* (salto condizionato)
- `if_ cmpge label` → se gli interi sul top dello stack sono **>=** allora va all'istruzione *label* (salto condizionato)
- `if_cmpgt label` → se gli interi sul top dello stack sono **>** allora va all'istruzione *label* (salto condizionato)
- `goto label` → va all'istruzione *label* (salto incondizionato)
- `ldc var` →

```
a = b * - c + b

iload b
iload c
ineg
imul
iload b
iadd
istore a
```

Tabella dei simboli

	Lex	Type	Address	Etc..
0	a	int	101…	…
1	b	int	110…	…
2	c	int	110…	…
3				
4				
5				

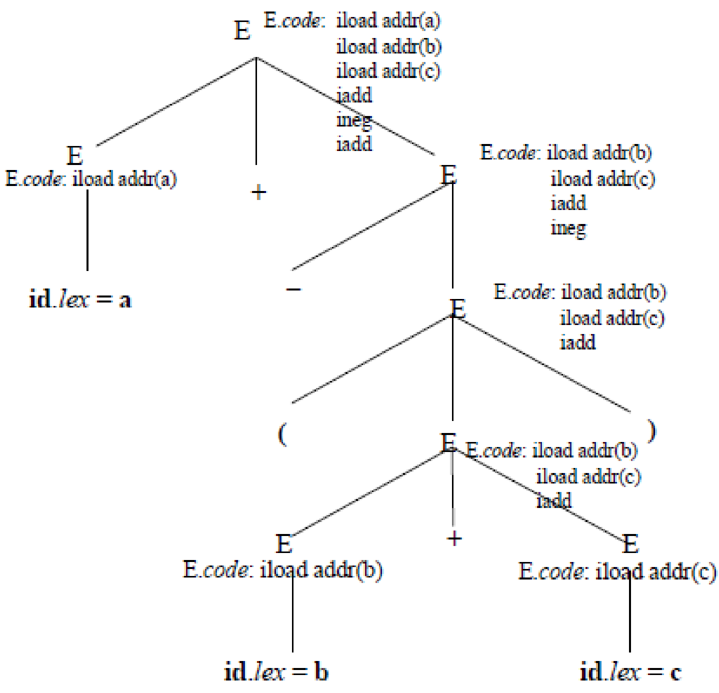
Una stringa con più di un albero di parsificazione ha di solito diversi significati quindi abbiamo bisogno di grammatiche non ambigue o, se si ha a che fare con grammatiche ambigue, di regole aggiuntive che risolvano l'ambiguità.
Pertanto assumeremo nel seguito, per le espressioni aritmetiche:

- + e * associativi a sinistra
- – con precedenza su *
- * con precedenza su +

Nelle regole semantiche per le espressioni aritmetiche:

- L'attr. sintetizz. **E.code** contiene il bytecode (codice postfisso)
- **addr(id.lex)** trova nella symbol table l'indirizzo associato al lessema
- **num.val** rappresenta il valore numerico del lessema num.
- **||** è l'operatore di concatenazione

Le traduzioni che definiamo sono delle sequenze di istruzioni postfisse (il bytecode), cioè delle stringhe.



$$
\begin{array}{ll}
E \rightarrow E_1 + E_2 & E.code = E_1.code||E_2.code||\boldsymbol{iadd} \\
E \rightarrow E_1 * E_2 & E.code = E_1.code||E_2.code||\boldsymbol{imul} \\
E \rightarrow -E_1 & E.code = E_1.code||\boldsymbol{ineg} \\
E \rightarrow (E_1) & E.code = E_1.code \\
E \rightarrow \boldsymbol{id} & E.code = \boldsymbol{iload\ addr(id.lex)} \\
\boldsymbol{E \rightarrow num} & \boldsymbol{E.code = ldc\ num.val}
\end{array}
$$

Traduzione di "a+- (b+c) "

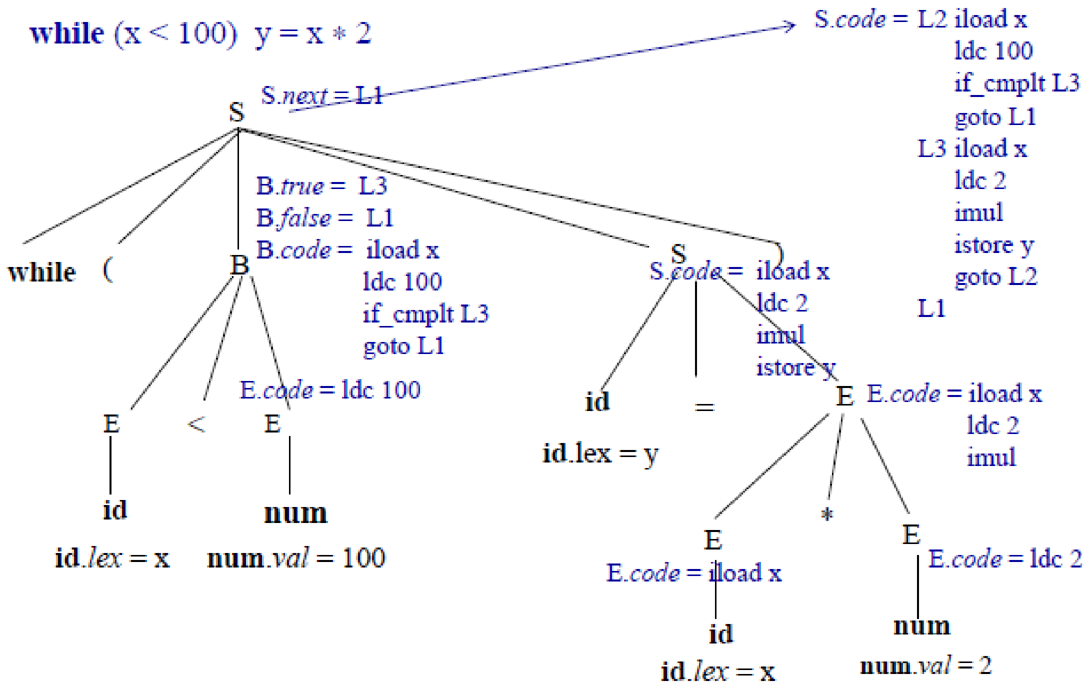
```
iload addr(a)        iadd
iload addr(b)        ineg
iload addr(c)        iadd
```

Statement

- Ad ogni **S** associamo, l'attributo sintetizzato **.code**, che contiene il codice postfisso e un attributo ereditato **.next** che contiene l'indirizzo (**label**) dell'istruzione successiva.
- **newlabel()** genera una nuova label simbolica
- **label(x.yyy)** indica il valore dell'attributo **x.yyy**

Espressioni booleane

- Ad ogni espressione booleana B vengono associati, oltre all'attributo sintetizzato .code che contiene il codice postfisso, due attributi ereditati .true e .false per definire le label delle prime istruzioni bytecode che traducono gli statement da eseguire nei casi B vero e B falso rispettivamente.
- $\boldsymbol{rel}.op \in \{<, \leq, >, \geq, ==, \neq\}$
- $\boldsymbol{cond}(\boldsymbol{rel})$ è una delle istruz. **if_cmpeq** $(=)$, **if_cmpne** $(\neq)$, **if_cmplt** $(<)$, **if_cmple** $(\leq)$, **if_cmpgt** $(>)$, **if_cmpge** $(\geq)$.
  Es: $\boldsymbol{cond}(=)$ è *if_cmpeq*.



Statement

$P \rightarrow S$	$S.next = newlabel(\,)$ $P.code = S.code \| \| label(S.next)$
$S \rightarrow id = E;$	$S.code = E.code \| \| istore(addr(id.lex))$
$S \rightarrow if(B)\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \| \| label(B.true) \| \| S_1.code$
$S \rightarrow if(B)\ S_1\ else\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \| \| label(B.true) \| \| S_1.code \| \| 'goto'\ S.next \| \| label(B.false) \| \| S_2.code$
$S \rightarrow while(B)\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \| \| B.code \| \| label(B.true) \| \| S_1.code \| \|\ 'goto'\ begin$
$S \rightarrow S_1\ ;\ S_2$	$S_1.next = newlabel(\,)$ $S_2.next = S.next$ $S.code = S_1.code \| \| label(S_1.next) \| \| S_2.code$
$S \rightarrow$ repeat S_1 until B (esci quando B è vera)	$begin = newlabel(\,)$ $S_1.next = newlabel(\,)$ $B.true\ =\ S_1.next()$ $B.false\ =\ begin$ $S.code = label(begin) \| \| S_1.code \| \| label(S_1.next) \| \| B.code$
$S \rightarrow \{S_1\}$	$S_1.next = S.next$ $S.code = S_1.code$

Espressioni booleane

$B \rightarrow E_1\ rel\ E_2$	$B.code = E_1.code \| \| E_2.code \| \| cond(rel.op) B.true \| \| 'goto'\ B.false$
$B \rightarrow true$	$B.code =$ ‘goto’ $B.true$
$B \rightarrow false$	$B.code =$ ‘goto’ $B.false$
$B \rightarrow !\,B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel(\,)$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| \| label(B_1.false) \| \| B_2.code$
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel(\,)$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| \| label(B_1.true) \| \| B_2.code$
$B \rightarrow (B_1)$	$B_1.true = B.true$ $B_1.false = B.false$ $B.code = B_1.code$

Espressioni aritmetiche

$E \rightarrow E_1 + E_2$	$E.code = E_1.code \| \| E_2.code \| \| iadd$
$E \rightarrow E_1 * E_2$	$E.code = E_1.code \| \| E_2.code \| \| imul$
$E \rightarrow -E_1$	$E.code = E_1.code \| \| ineg$
$E \rightarrow (E_1)$	$E.code = E_1.code$
$E \rightarrow id$	$E.code = iload\ addr(id.lex)$
$E \rightarrow num$	$E.code = ldc\ num.val$

La costruzione esplicita di lunghe stringhe di codice come valore degli attributi non è opportuna, tra l'altro, ad esempio, per il tempo richiesto per copiare le stringhe. In molti casi è possibile costruire incrementalmente porzioni di codice memorizzandole in un file.
Scriviamo lo schema di traduzione usando questa opzione. Sia $\boldsymbol{gen}$ la funzione che scrive sul file.

$E \to E_1 + E_2\ \{\boldsymbol{gen}(\boldsymbol{iadd})\}$
$E \to E_1 * E_2\ \{\boldsymbol{gen}(\boldsymbol{imul})\}$
$E \to -E_1\ \{\boldsymbol{gen}(\boldsymbol{ineg})\}$
$E \to (E_1)$
$E \to \boldsymbol{id}\ \{\boldsymbol{gen}(\boldsymbol{iload\ addr}(\boldsymbol{id.lex}))\}$
$\boldsymbol{E} \to \boldsymbol{num}\ \{\boldsymbol{gen}(\boldsymbol{ldc\ num.val})\}$
$B \to E_1\ \boldsymbol{rel}\ E_2\ \{\boldsymbol{gen}(\boldsymbol{cond}(\boldsymbol{rel.op})\ \boldsymbol{B.false})\}$

$P \to \{S.next = newlabel(\,)\}\ \boldsymbol{S}\ \{gen(label(S.next))\}$

$S \to \boldsymbol{id} = \boldsymbol{E}\ \{gen(istore(id.addr))\}$

$S \to \boldsymbol{if}\{B.true = newlabel(\,), B.false = S.next\}$
$(\boldsymbol{B})\ \{gen(label(B.true))\ \ S_1.next = S.next\}$
$\boldsymbol{S_1}\{gen(goto\ B.false)\}$

$S \to \boldsymbol{if}\{B.true = newlabel(\,), B.false = newlabel(\,)\}$
$(\boldsymbol{B})\{gen(label(B.true)), S_1.next = S.next\}$
$\boldsymbol{S_1}\ \boldsymbol{else}\ \{gen(label(B.false)), S_2.next = S.next\}\ \boldsymbol{S_2}$

$S \to \boldsymbol{while}\{begin = newlabel(\,), gen(label(begin)), B.true = newlabel(\,), B.false = S.next\}$
$(\boldsymbol{B})\{gen(label(B.true)), S_1.next = begin\}$
$\boldsymbol{S_1}\{gen(goto\ begin)\}$

$S \to \{S_1.next = newlabel(\,)\}\ \boldsymbol{S_1};\{S_2.next = S.next, gen(label(next))\}\ \boldsymbol{S_2}$

Gli schemi precedenti non sono realizzabili con il parser top-down deterministico che conosciamo: la grammatica, così come è scritta, è ambigua e non LL(1).
Si possono utilizzare gli schemi con qualche modifica alla grammatica e al parser.

Nel caso delle espressioni aritmetiche conviene usare la grammatica non ambigua ottenuta introducendo la variabile T (come usuale nelle espressioni).

$E \to E + T$
$E \to T$
$T \to -E$
$T \to (E)$
$T \to \boldsymbol{num}$
$T \to \boldsymbol{id}$

La funzione relativa a T è standard perché la produzioni hanno insiemi guida disgiunti.
Considerando la grammatica ottenuto $E \ \to\ E + T \mid T$ eliminando la ricorsione sinistra si ottiene una forma compatta per esprimere lo schema:

$E \to TE'$
$E' \to +T\ \{gen(\boldsymbol{iadd})\}\ E'$
$E' \to \varepsilon$

la grammatica sottostante è LL(1) e quindi lo schema è implementabile con discendente ricorsivo standard.

La tecnica più usata per realizzare schemi di questo tipo è però la seguente. Usando una notazione ibrida che combina la notazione delle grammatiche con quella delle espressioni regolari si potrebbe rappresentare lo schema per E in questo modo:

$$E \to T\ [\ + T\ \{\ gen(\boldsymbol{iadd})\ \}\ ]^*$$

che non è altro che una forma compatta per esprimere lo schema. Si può realizzare lo schema con la seguente funzione:

```
function E()
   begin T()
      while cc = '+'
         cc ← PROSS
         T()
         gen(iadd)
      return
   end
```

Note

- ε stringa vuota
- Φ linguaggio vuoto
- $|\varepsilon| = 0$
- $|\Sigma^0| = 1$
- $\Phi^* = \{\varepsilon\}$
- $\{\varepsilon\}^* = \{\varepsilon\}$
- L^+ può contenere ε se L contiene ε
- Condizione necessaria affichè una grammatica sia LL(1) è che non presenti ricorsione sinistra né prefissi comuni
- Esistono linguaggi regolari che non sono LR(0)? Si, poichè tutti i linguaggi che hanno intersezione non vuota con l'insieme dei propri prefissi non sono LR(0), mentre tutti i linguaggi finiti sono regolari. Il più semplice esempio è un linguaggio che contiene solo la stringa vuota ed un carattere.