



Fasi compilazione

Passaggio da un linguaggio ad alto livello a un di tipo macchina.

codice sorgente $\xrightarrow{\quad}$ codice oggetto \Rightarrow è un programma
 $\xrightarrow{\quad}$ altro linguaggio

compilatore \Rightarrow crea un eseguibile segnalando anche errori

interprete \Rightarrow prendendo il codice sorgente e tutte le volte eseguiamo istruzione per istruzione

compilatore Java: viene dato in pasto a un ibrido
traduttore che trasforma il sorgente in un nuovo linguaggio (Java Byte Code)
Ottenuto il programma intermedio viene interpretato da una macchina virtuale:
Java Virtual Machine.

⚠ I programmi interpretati sono molto più lenti rispetto a quelli compilati e cause delle traduzione istruzione per istruzione che perde di efficienza. Il compilatore ibrido però è meglio del traduttore classico, si posse prima per il linguaggio intermedio che si traduce più facilmente. Inoltre i programmi eseguiti nelle macchine virtuali sono più sicuri.

- 1) Processing: codice sorgente \Rightarrow codice sorgente modificato
- 2) Compilazione: il compilatore produce un codice assembly
- 3) Assembly: dall'assembly si posse al codice macchina (codice Rilasciato)
- 4) Linker / Loader: aggiunti i moduli delle librerie e file oggetto.

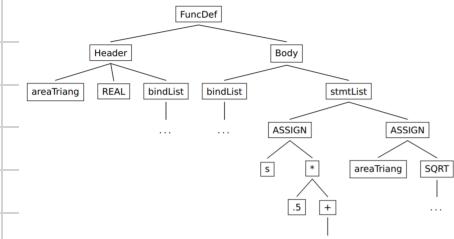


codice macchina
completo

Da linguaggio ad alto livello a linguaggio macchina

creazione di un albero sintattico estratto \rightarrow Fase di analisi

```
double areaTriang(double a, double b, double c) {
    double s = (a+b+c)/0.5;
    return Math.sqrt(s*(s-a)*(s-b)*(s-c));
}
```



Header: nome funzione, tipo del valore da restituire e liste di variabili
Body: lista variabili locali, pointer alle variabili successive (s)(stmtList), e successivamente una serie di istruzioni

Sintesi: trasformazione dell'albero in una serie di istruzioni in linguaggio macchina

Fase analisi

Analisi lessicale: 1^a fase compilazione eseguita da un modulo detto **scanner** che raggruppa i caratteri in sequenze elementari detti **lessemi**. Ad ogni lesseme viene assegnato una classe lessicale (**token**) e un attributo.

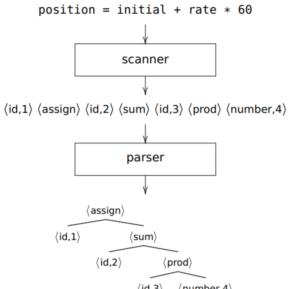
Per riuscire a leggere un programma, ricomponendone il contenuto (capire che position è un lesseme \rightarrow interpretato come variabile) si usa una teoria matematica: **Teorie degli automi a stati finiti**.

position = initial + rate * 60		
Lessema	Token	attributo
position	id	1
=	assign	-
initial	id	2
+	sum	-
rate	id	3
*	prod	-
60	number	4

tabella dei simboli

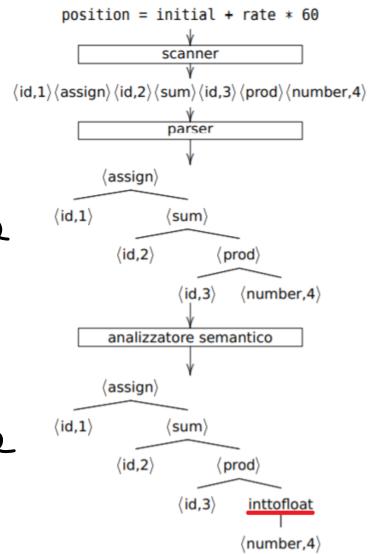
1	position
2	initial
3	rate
4	60

Analisi sintattica: l'analizzatore sintattico, il **parser** organizza i token generati dallo scanner in un albero: **albero sintattico estratto**.



Analisi semantica: usa sia la tabella dei simboli che l'albero sintattico estratto per controllare che il programma sia semanticamente coerente. Viene usato ad esempio il **type checking** che controlla se tutti gli operandi siano del tipo corretto. In C ad esempio è possibile eseguire una coercizione: ogni tipo ha le sue rappresentazioni in macchina e quindi se voglio fare una somma ad esempio devo prima fare un cast.

Questo analisi è statica perché rileva tutti gli errori che si verificano senza eseguire il programma. Altri errori possono essere quelli che si verificano a runtime.

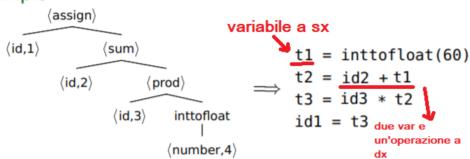


Fase sintetica

② Generazione codice intermedio: dall'albero estratto e dalle Tabelle dei simboli ottengo un linguaggio con istruzioni elementari e facile da tradurre.

→ indipendentemente dall'architettura (non vengono specificati indirizzi di memoria né registri in cui devono essere caricati i dati)

Esempio



variabile a sx

$$\begin{aligned}
 t1 &= \text{inttofloat}(60) \\
 t2 &= \underline{\underline{id2 + t1}} \\
 t3 &= \underline{\underline{id3 * t2}} \\
 id1 &= t3 \text{ due var e} \\
 &\text{un'operazione a dx}
 \end{aligned}$$

In questo caso viene creato un codice a 3 indirizzi: in tutte le istruzioni ho una variabile a sinistra e al massimo un'operazione e 2 variabili a destra.

③ Ottimizzazione: si cerca di ridurre il tempo e lo spazio necessari per l'esecuzione del codice intermedio. Questo miglioramento di efficienza non dipende ancora dell'architettura del sistema.

$$t1 = \text{inttofloat}(60)$$

$$t2 = id3 * t1$$

$$t3 = id2 + t2$$

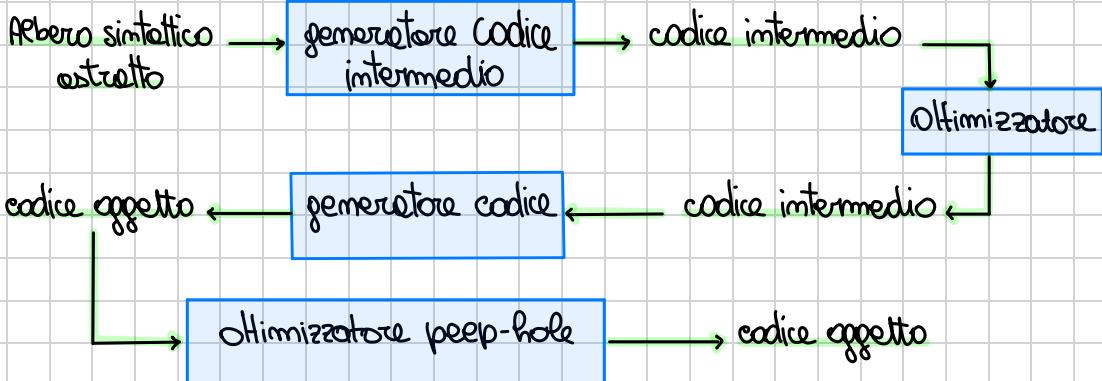
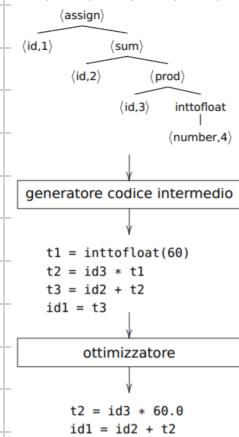
$$id1 = t3$$

$$\begin{aligned}
 t2 &= id3 * 60.0 \\
 id1 &= id2 + t2
 \end{aligned}$$

③ Generazione del codice oggetto: In questo step aviamo le vere e proprie traduzione in codice macchina.

- fissare le locazioni di memoria
- generare il codice per accedere a questi dati
- selezionare i registri per i calcoli intermedi
- ...

Queste operazioni dipendono strettamente dall'architettura in cui mi trovo. E quindi andrebbe ripetute per ciascuna architettura. Non dipende invece dal linguaggio ad alto livello di portante.

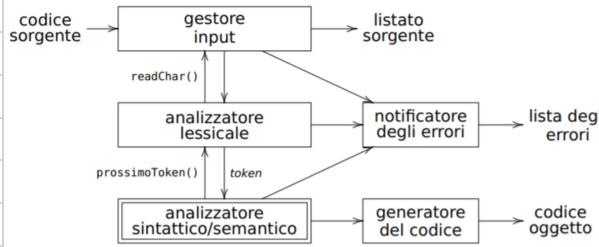


④ Otimizzazione Peep-Hole: ottimizzazione minore legata all'architettura.

Organizzazione del compilatore

Tutte queste fasi possono essere eseguite separatamente. Il nucleo del compilatore è l'analizzatore sintattico/semantico che attiva la generazione del codice come chiama funzioni opportune.

Un unico programma che invoca più funzioni.



Teoria dei linguaggi formali

Alfabeto: insiemi finiti non vuoti di simboli detti **lettere**

$$\Sigma_0 = \{a, b\}, \quad \Sigma_1 = \{0, 1\}, \quad \Sigma_2 = \{a, b, c\},$$

$$\Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

→ sequenze finite di questo alfabeto sono le **parole**

L'insieme delle parole su Σ è Σ^*

↳ Le parole che non contiene alcuna lettera si chiama **parola vuota**, è denotata con ϵ

concatenazione: operazione binaria (prendere 2 parole in input), e totale (quando le 2 parole la concatenazione è definita)

Gode delle seguenti proprietà

- per ogni $u, v, w \in \Sigma^*$, $(uv)w = u(vw)$ (proprietà associativa),
- per ogni $u \in \Sigma^*$, $u\epsilon = \epsilon u = u$ (elemento neutro),
- se si ha $uw = vw$ oppure $wu = vw$ con $u, v, w \in \Sigma^*$, allora $u = v$ (cancellatività a destra e a sinistra).

fattore: sequenza di lettere consecutive all'interno delle parole.

Se la prima lettera della parola è vuote il fattore è detto **prefisso**, se ad essere vuote è l'ultima lettera allora si dice **suffisso**.

Nel caso in cui il fattore è diverso dalle parole stesse, è detto **fattore proprio**.

Definizione

Diremo che una parola v è un **fattore** di una parola w se risulta $w = xv$ per opportune parole x, y . Nel caso in cui $x = \epsilon$ (risp., $y = \epsilon$) il fattore v si dice **prefisso** (risp., **suffisso**) di w . Diremo che v è un fattore **proprio** se $v \neq w$.

Linguaggio formale è ogni sottoinsieme Σ^* di Σ , un insieme di parole. Se è finito posso elencarne gli elementi ed uscirlo facilmente. Se è infinito non li posso elencare ed è più difficile capire se un elemento me faccia parte o meno. Per descrivere un linguaggio infinito in parte, è una sequenze di simboli presi da un linguaggio finito. Definito un linguaggio è possibile scrivere una parola. È difficile invece scrivere insiemi infiniti di parole.

Linguaggio infinito: dato in pasto ad un compilatore non genera errori

Teorema di Cantor: Se è possibile definire qualche linguaggio infinito, il teorema di Cantor ci dice che non è possibile definirli tutti. Non ci sta una funzione che ad ogni linguaggio sull'alfabeto Σ , mi associa una parola sull'alfabeto Σ .

Grammatica e struttura di frase

(e)

→ È una quadrupla: $G = \langle V, \Sigma, P, S \rangle$ dove:

- V = vocabolario totale (alfabeto finito di simboli)
- $\Sigma \subseteq V$ = simboli terminali
- P = insieme delle produzioni
- S = simbolo iniziale

- una parola β è una **conseguenza diretta** di α se in α sostituisco un fattore che è il lato sinistro di una produzione con il lato destro.
- una **conseguenza** (non per forza diretta) è una catena di conseguenze dirette che va da α a β .
- le **forme sentenziali** sono le parole **derivate dal simbolo iniziale**.

(e)

Se una forma sentenziale non contiene variabili allora è una parola del **linguaggio generato dalla grammatica**.

↳

insieme di tutte le parole costituite solo da termini ottenibili partendo dal simbolo iniziale tramite produzioni

Due grammatiche si dicono equivalenti se generano lo stesso linguaggio

Riconoscimento: capire se una parola appartiene a un linguaggio. Serve per capire se il nostro codice è sintatticamente corretto. Capisco se il codice usa parole appartenenti al linguaggio e alle grammatiche.

(e)

→ prendo in input una grammatica e una parola determinata sul linguaggio della grammatica. Come output ha un booleano in base a se è generata dalla grammatica oppure no.

Parsim: prende come input una grammatica, una parola del linguaggio generato, e vuole trovare una derivazione delle parole, cioè una sequenza di forme semantici ognuna conseguenza diretta delle precedenti, in modo che a partire dal simbolo iniziale otenga la parola.

⚠ Non esiste un algoritmo generale che risolva riconoscimento o parsim per tutte le grammatiche. Bisogna quindi restringere la classe di grammatiche.

Efficienza vs Espressività

- ⇒ Grammatiche con produzioni semplici: parsim e riconoscimento efficienti ma linguaggi poco espressivi
- ⇒ Grammatiche più generali. Linguaggi potenti ma algoritmi lenti o inesistenti

Classificazione delle produzioni

La strategia è quindi classificare le grammatiche in base alle forme delle produzioni: quelle con una forma più semplice avranno una riconoscimento e parsim semplici e veloci con però linguaggi meno espressivi. Fino a linguaggi più complessi e generali con riconoscimento e parsim difficili o impossibili.

Gerarchie di Chomsky: divide le grammatiche e i linguaggi generati da esse in 4 tipi

- **Linguaggi di tipo 0**: appartengono a questa tipologia le grammatiche e strutture di frase. I linguaggi generati da queste grammatiche si dicono linguaggi di tipo 0 o ricorsivamente enumerabili.
Se ho una procedura infinita che mi genera una lista di parole, quella lista di parole è un linguaggio di tipo zero.
- **Linguaggi di tipo 1**: (Grammatiche contestuali) Sono linguaggi sensibili al contesto. Ad esempio date le parole: xAy , la grammatica può dire sostituisco la lettera A con la B solo se A è tra x e y. Le nuove frasi non possono essere più courti di quelle vecchie, sono dunque uguali oppure più lunghe.

→ **Grammatiche monotonie**: simili alle grammatiche contestuali ma con il solo vincolo di essere delle stesse lunghezze o maggiore. Non serve guardare il contesto.

- **Linguaggi di tipo 2**: (Grammatiche non contestuali) Sono le grammatiche più usate nella programmazione. Non hanno alcuna dipendenza dal contesto. Ad esempio considero le produzioni: $A \rightarrow \text{qualsiasi}$, A è una variabile, un simbolo non terminale, "qualsiasi" può essere lettere, altri simboli o una combinazione. Sono una sottoclasse dei linguaggi di tipo 1.

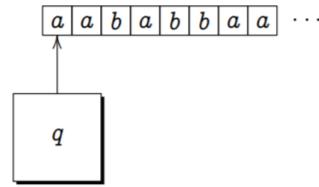
→ Sono accettati da automi a pila ($a^m b^n \mid m \geq n$)

- **Linguaggi di tipo 3**: le produzioni sono ancora più semplici cioè destra-lineari: $A \rightarrow aB$ o $A \rightarrow a$, sinistra-lineari: $A \rightarrow Ba$ o $A \rightarrow a$. I linguaggi di tipo 3 sono una sottoclasse di quelli di tipo 2, un linguaggio di tipo 3 è anche di tipo 2 ma non sempre contrario. Parsing e riconoscimento molto efficienti, importanti per l'analisi lessicale (dei token che formano il codice)

Teoria automi a stati finiti

Un automo a stati finiti è una macchina che legge una parola, lettera per lettera e decide se quella parola va bene oppure no. Immagina una macchina che raggiunge un numero finito di configurazioni interne.

Parte da uno stato iniziale e legge le prime lettere delle parole. A seconda delle lettere lette passa ad un nuovo stato e così via per ogni lettera. Se alla fine raggiungo uno stato finale la parola è accettata, altrimenti no.



Algoritmo lineare, e ogni cella letta so se tutto ciò che è venuto prima è stato accettato o meno.

Un automo deterministico è una quintupla $A = \langle Q, \Sigma, \delta, q_0, F \rangle$

Q = insieme degli stati, Σ = alfabeto in input, δ = funzione di transizione,
 q_0 = stato iniziale, F = insieme degli stati finali

Le funzione δ di transizione è le regole che dice: "Se sono sullo stato q e leggo la lettera a , allora vado nello stato p "

$$\Rightarrow \delta(q, a) = p$$

Per estendere la funzione di transizione in modo che non lavori solo con le coppie stato-lettera ma anche con coppie stato-parola, introduco $\widehat{\delta}$.

cappellino per far capire che la funzione delta è stata estesa
 La funzione δ si estende a $\widehat{\delta} : Q \times \Sigma^* \rightarrow Q$ ponendo

$$\begin{aligned} \widehat{\delta}(q, \varepsilon) &= q, && \text{per ogni } q \in Q, \\ \widehat{\delta}(q, va) &= \widehat{\delta}(\widehat{\delta}(q, v), a), && \text{per ogni } q \in Q, v \in \Sigma^*, a \in \Sigma. \end{aligned}$$

ultima lettera della parola

Calcoliamo $\widehat{\delta}(q_0, aabb)$ per l'automa dell'esempio precedente:

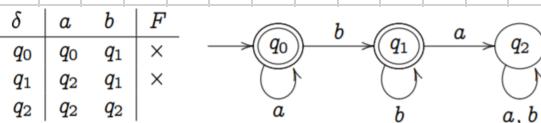
$$\begin{aligned} \widehat{\delta}(q_0, \varepsilon) &= q_0, \\ \widehat{\delta}(q_0, a) &= \widehat{\delta}(\widehat{\delta}(q_0, \varepsilon), a) = \widehat{\delta}(q_0, a) = q_0, \\ \widehat{\delta}(q_0, aa) &= \widehat{\delta}(\widehat{\delta}(q_0, a), a) = \widehat{\delta}(q_0, a) = q_0, \\ \widehat{\delta}(q_0, aab) &= \widehat{\delta}(\widehat{\delta}(q_0, aa), b) = \widehat{\delta}(q_0, b) = q_1, \\ \widehat{\delta}(q_0, aabb) &= \widehat{\delta}(\widehat{\delta}(q_0, aab), b) = \widehat{\delta}(q_1, b) = q_1. \end{aligned}$$

	a	b
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

$Q = \{q_0, q_1, q_2\}$
 $\Sigma = \{a, b\}$
 $F = \{q_0, q_1\}$

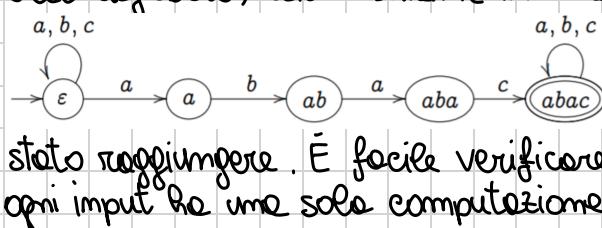
L'insieme delle parole accettate dall'automa A si dice linguaggio riconosciuto e si denota con $L(A)$.

Per capire meglio il comportamento di un automa si crea un grafo diretto con frecce etichettate, i vertici rappresentano gli stati dell'automa e le frecce sono le triple costituite da stato di partenza, stato di destinazione ed etichetta.



Automi e stati finiti deterministici

È un automa in cui per ogni stato q e per ogni simbolo a dell'alfabeto, esiste esattamente una transizione definita $\delta(q, a)$.



Il comportamento è completamente deterministico, non ci sono ambiguità su quale stato raggiungere. È facile verificare se una parola è accettata, ogni input ha una sola computazione possibile.

Automi e stati finiti non deterministici

È un automa dove, dato uno stato q e un simbolo a possono esserci:

- Zero, uno o più transizioni
- Transizioni ϵ (passaggi di stato senza consumare simboli)

Caratteristiche: È possibile avere più computazioni parallele, per cui una parola è accettata se almeno una computazione porta ad uno stato finale. Una parola è rifiutata se tutte le computazioni finiscono in stati non finali, o si bloccano prima che l'input sia terminato.

Un autome a stati finiti non deterministico è sempre una quintupla:

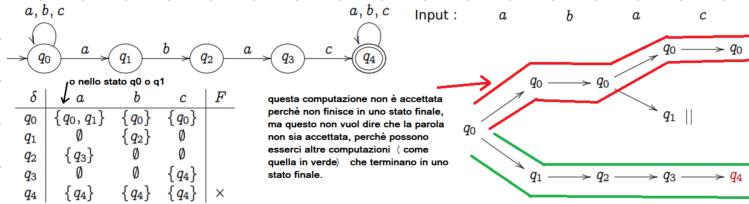
Q = insieme degli stati finiti

Σ = alfabeto di input

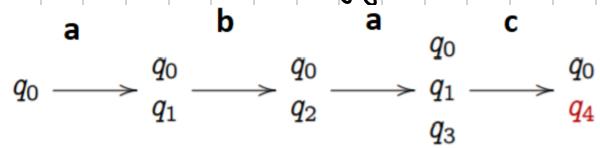
δ = funzione di transizione

$q_0 \in Q$ = stato iniziale

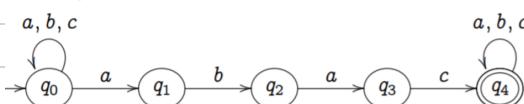
$F \subseteq Q$ = insieme degli stati finali



NB Se A è un autome a stati finiti non deterministico, esiste effettivamente un equivalente autome a stati finiti deterministico A' tale che $L(A) = L(A')$, cioè esiste l'autome deterministico e l'algoritmo per costruire questo autome a partire dall'autome non deterministico. Anche qui quindi in base alle configurazioni delle macchine e alle lettere im input, determiniamo le configurazioni successive. In questo caso però ho delle colomme di stati.

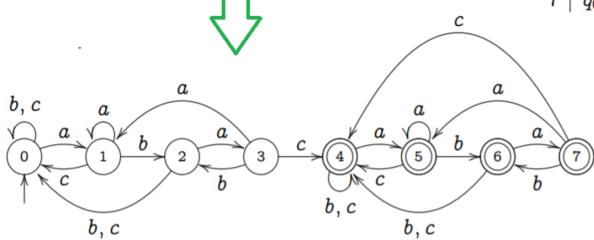


Esempio



i	s_i	$\delta'(i, a)$	$\delta'(i, b)$	$\delta'(i, c)$	$i \in F$
0	q_0	1	0	0	
1	q_0, q_1	1	2	0	
2	q_0, q_2	3	0	0	
3	q_0, q_1, q_3	1	2	4	
4	q_0, q_4	5	4	4	
5	q_0, q_1, q_4	5	6	4	
6	q_0, q_2, q_4	7	4	4	
7	q_0, q_1, q_3, q_4	5	6	4	

con la lettera b: da q0 con b vado in q0, da q1 con b vado in q2.



Algoritmo di determinizzazione :

Si usa una lista di stati che viene inizializzata solo con lo stato q_0 , per ogni stato τ delle liste e ogni lettera $a \in \Sigma$, calcoliamo $s = \delta(\tau, a)$. Se otengo un nuovo stato lo aggiungo alla lista. Calcolo anche l'insieme F' degli stati finali.

Algoritmo 1: Determinizzazione

Ingresso: Un automa non deterministico $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$

Uscita: Un automa deterministico $\mathcal{A}' = \langle Q', \Sigma, \delta', s_0, F' \rangle$ equivalente

$s_0 \leftarrow \{q_0\};$

$n \leftarrow 1;$ // numero degli stati

$j \leftarrow 0;$

$F' \leftarrow \emptyset;$

mentre $j < n$ **fai**

per ciascun $a \in \Sigma$ **fai**

$s \leftarrow \bigcup_{q \in s} \delta(q, a);$

se $s = s_i$ per qualche $i < n$ **allora**

$\delta'(j, a) \leftarrow i$

altrimenti

$s_n \leftarrow s;$

$\delta'(j, a) \leftarrow n;$

se $s \cap F \neq \emptyset$ **allora** $F' \leftarrow F' \cup \{n\};$

$n \leftarrow n + 1$

$j \leftarrow j + 1$

Automa e stati finiti non deterministico con ε transizioni

Si parla sempre di una quintupla del tipo $A = \langle Q, \Sigma, S, q_0, F \rangle$
 Q insieme finito di stati

Σ alfabeto

δ funzione di transizione che permette transizioni su ϵ

$q_0 \in Q$, stato iniziale

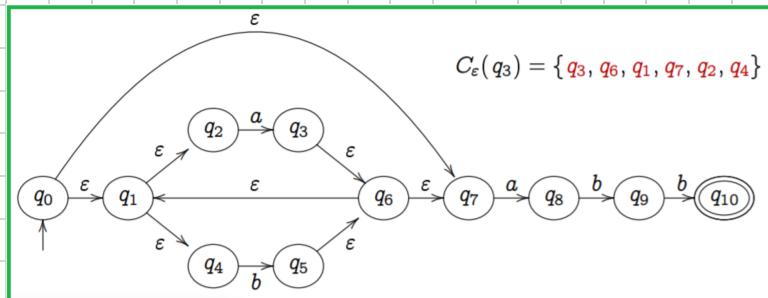
$F \subseteq Q$, insieme degli stati finali

Una parola $w \in \Sigma^*$ è accettata dall'automa se esiste almeno un cammino che parte da q_0 , termine in uno stato di F e legge esattamente i simboli della parola w , con le possibilità di fare transizioni su ϵ (spostamenti tra stati senza consumare caratteri delle parole)

→ **ε-chiuse**: di uno stato q , indicata con ϵ -chiura (q), è l'insieme di tutti gli stati raggiungibili da q usando solo ε transizioni (anche più di una concatenata)

Per calcolare le ϵ chiuse di uno stato si può usare questo algoritmo:

- 1) Listi che contiene solo q : $R \leftarrow (q)$
- 2) Inizializzo puntatore che punta al primo elemento della lista R : $p \leftarrow$
- 3) While $p \neq \text{Nil}$
 - 4) prendo tutti gli elementi che si ottengono partendo dal primo elemento della lista e seguendo le ϵ transizioni le aggiungiamo alla lista, controllando che non sono già presenti.
 - a. append ($R, S(p, \epsilon)$)
 - b. $p \leftarrow$ successivo (P)
- 5) return R



Sia $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ l'automa non deterministico con ϵ -transizioni.

Definisco l'automa deterministico $\mathcal{A}' = \langle Q', \Sigma, \delta', s_0, F' \rangle$ come segue:

- l'insieme degli stati è l'insieme $Q' = \wp(Q)$ costituito dai sottoinsiemi di Q ,
- lo stato iniziale è $s_0 = C_\epsilon(q_0)$,
- gli stati finali sono tutti i sottoinsiemi di Q che contengono almeno un elemento di F , cioè

$$F' = \{s \in \wp(Q) \mid s \cap F \neq \emptyset\},$$

Calcolo delta' di s,a dove

s =stato del mio nuovo A det., cioè un'insieme di stati dell'A di partenza. a = una qualunque lettera

la funzione di transizione $\delta': Q' \times \Sigma \rightarrow Q'$ è definita da

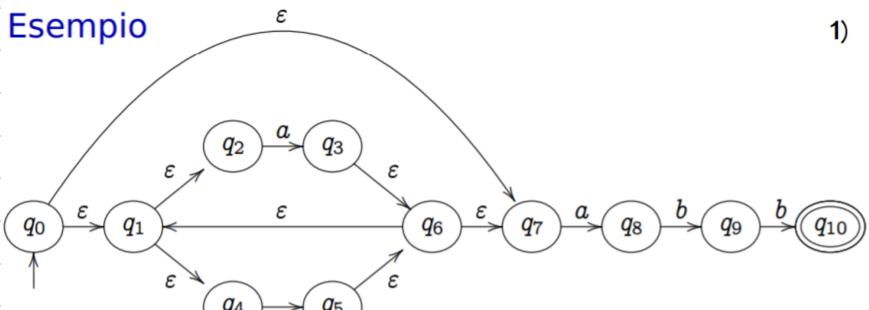
$$\delta'(s, a) = C_\epsilon \left(\bigcup_{p \in s} \delta(p, a) \right), \quad s \in \wp(Q), a \in \Sigma.$$

L'insieme degli stati è costituito dalle parti dell'insieme degli stati dell'automa di partenza. Cioè uno stato del nuovo A deterministico è un insieme di stati dell'A non det.

gli stati finali sono tutti quei sottoinsiemi che contengono almeno uno stato finale dell'A precedente.

In sostanza, $\delta'(s, a)$ sarà l'insieme di tutti quegli stati che posso raggiungere nel vecchio A partendo da uno qualunque degli stati di s e poi seguendo la freccia con etichetta a , e poi tutte le frecce che voglio con etichetta Epsilon.

Esempio



i	$C_\epsilon(q_i)$
0	$\{q_0, q_1, q_2, q_4, q_7\}$
1	$\{q_1, q_2, q_3, q_4, q_6, q_7, q_8\}$
2	$\{q_1, q_2, q_4, q_5, q_6, q_7\}$
3	$\{q_1, q_2, q_4, q_5, q_6, q_7, q_9\}$
4	$\{q_1, q_2, q_4, q_5, q_6, q_7, q_{10}\}$
5	$\{q_1, q_2, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}\}$
6	$\{q_1, q_2, q_4, q_6, q_7\}$
7	$\{q_7\}$
8	$\{q_8\}$
9	$\{q_9\}$
10	$\{q_{10}\}$

2)

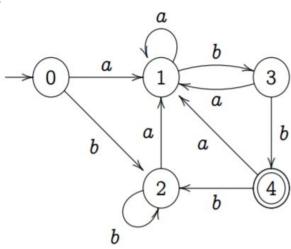
j	s_j	$\delta'(j, a)$	$\delta'(j, b)$	$j \in F$
0	$\{q_0, q_1, q_2, q_4, q_7\}$	1	2	
1	$\{q_1, q_2, q_3, q_4, q_6, q_7, q_8\}$	1	3	
2	$\{q_1, q_2, q_4, q_5, q_6, q_7\}$	1	2	
3	$\{q_1, q_2, q_4, q_5, q_6, q_7, q_9\}$	1	4	
4	$\{q_1, q_2, q_4, q_5, q_6, q_7, q_{10}\}$	1	2	X

stato chiusura di q_3 e q_8 , che corrisponde a j_1 .
 q_3 e q_8 perché con 'a' partendo da j_0 , posso arrivare a questi due stati.

Devo seguire da ciascuno di questi 5 stati le frecce etichettate a. Gli unici stati che hanno frecce uscenti 'a', sono q_2 e q_7 . Da q_2 vado in q_3 , e da q_7 vado in q_8 . Ora si fa la Epsilon chiusura di queste coppie di stati. q_8 non ha frecce uscenti etichettate Epsilon, quindi la sua chiusura si ferma a q_8 . q_3 invece contiene $q_3, q_6, q_7, q_1, q_2, q_4$.

Dobbiamo quindi aggiungere alla nostra lista di stati, quelli nuovi che abbiamo trovato: q_3, q_6, q_8 .

Infine controllo se lo stato è finale, ma dato che non contiene q_{10} , non lo è.



→ graph dell'esempio

Espresioni Regolari

Sempre per costruire e rappresentare oggetti infiniti con una descrizione finita.

Per costruire nuovi linguaggi a partire da quelli noti possiamo applicare le seguenti operazioni.

- unione
- intersezione
- complemento
- concretazione
- potenza : concretazione di n coppie di un linguaggio con se stesso
- chiusura di Kleene : prendo tutte le potenze di un linguaggio e me ne faccio l'unione.

{ e differenze delle altre operazioni che applicate a insiemi finiti restituiscono insiemi finiti, la chiusura di Kleene restituisce uno infinito e meno l'insieme vuoto e comunque solo le parole vuote

Operazioni normali su linguaggi finiti

Chiusura di Kleene su linguaggi infiniti

N.B per definire i linguaggi infiniti, basta che specifico le operazioni fatte per ottenerli : **espressioni regolari**

Definizione : sia $\hat{\Sigma}$ l'alfabeto ottenuto aggiungendolo a Σ : $\emptyset, +, *, (,$) . Si dicono espressioni regolari sull'alfabeto Σ le parole sull'alfabeto $\hat{\Sigma}$ che si ottengono applicandone un numero finito di volte le seguenti regole :

- i) Ogni lettera $a \in \Sigma$ è un'espressione regolare, \emptyset è una regex
- ii) Se E e F sono espressioni regolari, allora lo sono: $(A+F), (AF), E^*$

A ogni espressione regolare è associato un linguaggio, detto linguaggio denotato da un'espressione regolare, con queste regole:

- $\forall a \in \Sigma$, l'espressione regolare a denota il linguaggio $\{a\}$;
- l'espressione regolare \emptyset denota il linguaggio vuoto.

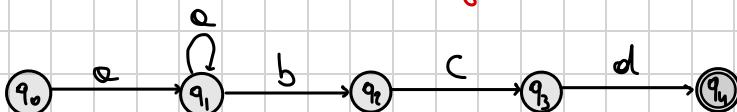
- Detti L_E e L_F i linguaggi denotati dalle espressioni regolari E e F , i linguaggi denotati dalle espressioni regolari $(E+F)$, (EF) , E^* sono rispettivamente, $L_E \cup L_F$, $L_E L_F$, L_E^*

Teorema di Kleene: un linguaggio è regolare se e solo se è riconosciuto da un autome e stati finiti

Sintesi: esiste un algoritmo che a partire da un'espressione regolare, produce un'autome e stati finiti che accetta il linguaggio denotato da tale espressione

Analisi: dato un autome e stati finiti, questo produce un'espressione regolare che denota il linguaggio accettato da tale autome. Dalle macchine ottempo la descrizione del comportamento

↓
Dato' autome all'espressione
regolare



Definisco L_{ijk} come i cammini dal q_i a q_j che passano solo per $q_k \leq k$

L_{043}

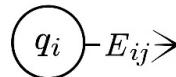
$$\left. \begin{array}{l} q_0 \rightarrow a q_1 \\ q_1 \rightarrow a^* q_1 \\ q_1 \rightarrow b q_2 \\ q_2 \rightarrow c q_3 \\ q_3 \rightarrow d q_4 \end{array} \right\} \Rightarrow a \cdot a^* b \cdot c \cdot d = a^* bcd$$

Metodo di eliminazione degli stati

- 1 selezioniamo uno stato q_k che non sia né iniziale né finale;
- 2 per ogni freccia $q_i \xrightarrow{E_{ik}} q_k$ che entra nello stato q_k e ogni freccia $q_k \xrightarrow{E_{kj}} q_j$ che esce da tale stato ($i, j \neq k$)
 - 1 rimpiazziamo l'etichetta della freccia $q_i \xrightarrow{E_{ij}} q_j$ con $E_{ij} + E_{ik} E_{kk}^* E_{kj}$ (o $E_{ij} + E_{ik} E_{kj}$ qualora manchi la freccia da q_k a q_k);
 - 2 se tale freccia è assente, la creiamo (con etichetta $E_{ik} E_{kk}^* E_{kj}$, ovvero $E_{ik} E_{kj}$);
- 3 rimuoviamo lo stato q_k e tutte le frecce che entrano o escono da esso;
- 4 ripetiamo la procedura fino a ottenere un automa con due stati e una sola freccia;
- 5 l'etichetta di tale freccia è l'espressione regolare cercata.

PASSI PER TROVARE UN'ESPRESSONE REGOLARE

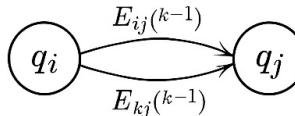
1. Trova E_j , cioè le etichette dirette fra ogni coppia di stati.



2. Elimina uno stato alla volta usando la formula

$$E_{jk} = E_{ij}^{(k-1)} + E_{ik}^{(k-1)} (E_{kk}^{(k-1)} E_{kj}^{(k-1)})$$

3. Ripeti finché rimane solo il nodo iniziale e il nodo finale



L'etichetta sull'arco tra di loro è $E_{ij}^{(n)}$

Autome Mimimo

Autome deterministico con il minore numero di stati

→ **Autome di Nerode**: questo autome di un linguaggio regolare L è un autome deterministico che accetta L col minimo numero di stati possibile.

Si guarda quali parole si comportano allo stesso modo, ovvero le parole che possono essere complete con le stesse lettere, per ottenere parole di L .

equivalemza: ogni classe raccolge parole che sono equivalenti secondo Nerode, tutte le parole che aggiungero qualcosa restano nello stesso linguaggio, devono essere nello stesso gruppo.
Ogni classe sarà uno stato dell'automa mimimo

costruzione: creo uno stato per ogni classe di equivalenza, lo stato iniziale è quello che contiene la parola vuota. Gli steti finali sono le classi che contengono parole di L . Le transizioni sono costruite con la concatenazione di lettere dell'alfabeto.

congruenza destra

→ Se 2 parole sono equivalenti, allora aggiungendo le stesse lettere a destra, restano equivalenti.

es. $M = "a"$, $V = "b"$ con " a " ≈ " b "; se è una congruenza destra allora aggiungendo una stessa lettera a destra le parole restano equivalenti. ($Ma = "aa"$, $Va = "ba" \Rightarrow aa \approx ba$)

Equivalemza di Nerode

cosa e? \rightsquigarrow un modello per dire quando 2 parole si comportano allo stesso modo rispetto un linguaggio L .

definizione: Dati un linguaggio L su un alfabeto Σ e due parole u, v su Σ^* , si dice che u è equivalente a v se:

A parole y vale $\rightsquigarrow uy \in L \iff vy \in L$. Quindi se posso completare u, v con le stesse parole per ottenere qualcosa che appartiene a L



questo è alla base delle costruzioni di un autome minimmo. Ogni classe di equivalenza (gruppo con parole equivalenti) diventa uno stato dell'autome.

Proprietà

- **Riflessiva** \rightarrow ogni parola è equivalente a se stessa ($u \approx u$)
- **Simmetrica** \rightarrow se $u \approx v \Rightarrow v \approx u$
- **Transitiva** \rightarrow se $u \approx v$ e $v \approx w \Rightarrow u \approx w$

la relazione N_L è una congruenza destra (se $u \approx v \Rightarrow ua \approx va, \forall a$)

Dimostrazione

N_L è un'equivalenza.

Dobbiamo verificare che per ogni $u, v \in \Sigma^*$, $a \in \Sigma$, se $u N_L v$, allora $ua N_L va$.

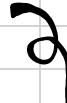
Invero sia $u N_L v$ e $a \in \Sigma$.

Dalla definizione di N_L si ha $uy \in L \iff vy \in L$ per ogni $y \in \Sigma^*$

Sostituendo y con ay , ottengo $uay \in L \iff vay \in L$

Quindi si ha anche $ua N_L va$.

L è unione di classi di equivalenza di N_L . Ogni classe di equivalenza contiene parole che si comportano allo stesso modo. L'insieme Σ^* è diviso in classi disgiunte (senza elementi in comune). Il linguaggio L è unione di intere classi: o una classe sta dentro L o tutte fuori, non può stare in metà.



Dimostrazione

Sia $u \in L$, $v \in L \iff uy \in L \iff vy \in L$ per ogni $y \in \Sigma^*$.

Prendendo $y = \epsilon$ ottengo $u \in L \iff v \in L$.

Quindi ogni classe di equivalenza o sta tutta dentro L o sta tutta nel complemento.

Teorema di Myhill-Nerode

Sia L un linguaggio su Σ , le seguenti proposizioni sono equivalenti:

- i) L è regolare
- ii) L è unione di classi di una congruenza destra su Σ^* di indice finito
- iii) N_L ha indice finito (\Rightarrow indice = numero classi di equivalenze dell'equivalenza stessa)

Per dimostrare che queste 3 condizioni (sono equivalenti), uso una dimostrazione circolare: se è vero i è vero anche ii e se è vero ii allora anche iii e poi i.

i \Rightarrow ii

Idea: se L è accettato da un automa deterministico, allora posso costruire una partizione di Σ^* in base agli stati del DFA

costruzione:

- Sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA che accetta L
- per ogni stato $q \in Q$ definisco l'insieme:
 $L_q = \{w \in \Sigma^* \mid \delta(q_0, w) = q\}$

Gli insiemi L_q formano una partizione di Σ^* : ogni parola conduce ad un solo stato

- Definisco una relazione \sim su Σ^* tale che: $u \sim v \iff \delta(q_0, u) = \delta(q_0, v)$
- Questa è una congruenza destra: $u \sim v \Rightarrow uq \sim vq \quad \forall q \in Q$

L'indice di \sim è al massimo $|Q|$, quindi finito

ii \Rightarrow iii

Se esiste una congruenza destra di indice finito che definisce L , allora la relazione di Nerode N_L ha indice finito

Definizioni:

- una congruenza destra \sim è una relazione di equivalenza compatibile a destra: $u \sim v \Rightarrow u\alpha \sim v\alpha$
- La relazione di Nerode è: $uN_L v \Leftrightarrow \forall z \in \Sigma^*, uz \in L \Leftrightarrow vz \in L$

Dimostrazione:

- suppongo di avere una congruenza \sim con indice finito e L unione di alcune classi
- prendiamo 2 parole $u N_L v$
- dimostro che per ogni classe delle Nerode è contenuta in un'unione di classi \sim
- quindi l'indice di N_L è minore o uguale a quello di \sim e quindi finito

iii \Rightarrow i

Se N_L ha indice finito, allora L è regolare

\rightsquigarrow procedo costruendo un automa finito deterministico basato sulle classi di equivalenza delle Nerode

costruzione automa di Nerode A :

- stati: le classi di equivalenza $[w]_{N_L}$
- stato iniziale: la classe $[\epsilon]$ delle parole vuote
- transizioni: $\delta([w], Q) = [wQ]$
- stati finali: $F = \{[w] \mid w \in L\}$

\rightsquigarrow Se una parola w porta a uno stato finale $\Rightarrow w \in L$

\rightsquigarrow Poiché l'indice di N_L è finito, l'automa ha un numero finito di stati

\rightsquigarrow Conclude che il linguaggio accettato è proprio $L \rightarrow L$ è regolare

L regolare \Leftrightarrow esiste una congruenza destra di indice finito $\Leftrightarrow N_L$ ha indice finito

Un linguaggio è regolare se e solo se induce un numero finito di comportamenti distinguibili sul futuro delle parole

L'automa di Nerode è l'automa deterministico che accetta L con il minor numero possibile di stati

Dimostrazione:

- Sia A un DFA che accetta L con numero minimo di stati, diciamo m
 - Già dimostrato con teorema Nerode:
 - L è unione di classi di una congruenza destra \sim con indice m
 - L'indice della relazione di Nerode N_L è minore o uguale all'indice di qualunque congruenza destra che definisce L , quindi indice $(N_L) \leq m$
 - Nella direzione opposta:
 - Si costruisce un DFA che accetta L usando le classi di N_L . Questo automa è un numero di stati pari all'indice di N_L .
- ⇒ A ha almeno tanti stati quante sono le classi di N_L , quindi l'automa di Nerode è minimo.

Strategie:

- considero solo gli stati accessibili
- gli stati dell'automa minimo corrispondono alle classi della relazione N_L
- Queste classi corrispondono a sottosetimi di stati di Q : N_L induce una partizione di Q

Le condizioni che questa partizione deve rispettare sono:

- 1) l'insieme F degli stati finali è unione di classi
- 2) Se 2 stati p e q sono nella stessa classe, allora per ogni simbolo a , anche $\delta(p, a)$ e $\delta(q, a)$ devono stare nella stessa classe.
- 3) La partizione deve essere la più fine possibile (minore numero di classi che soddisfano 1 e 2)

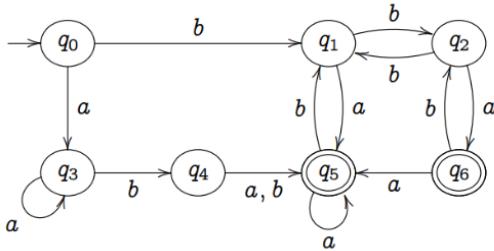
Algoritmo di minimizzazione (basato su raffinamenti successivi)

- 1) inizio con una partizione iniziale $\Pi_0 = \{F, Q \setminus F\}$
- 2) costruisco le partizioni successive Π_1, Π_2, \dots nel seguente modo:
 - In ogni passaggio Π_{m+1} si ottiene da Π_m spezzando le classi che non rispettano la condizione 2 (transizioni coerenti)
- 3) Mi fermo quando $\Pi_{m+1} = \Pi_m$, ovvero non si possono più spezzare le classi

Costruzione dell'automa minimo (ottenute le transizioni finali):

- Gli stati sono le classi della partizione Π
- per ogni classe C e ogni lettera $a \in \Sigma$, $\delta'(C, a)$ è la classe che contiene tutti gli stati $\delta(q, a)$ con $q \in C$
- lo stato iniziale è la classe C_0 che contiene lo stato q_0 di A
- gli stati finali sono le classi contenute in F

ESEMPIO Voglio minimizzare il seguente automa deterministico.



Innanzitutto divido l'insieme degli stati in due classi: stati finali e stati non finali.

$$\Pi_0 = (Q \setminus F, F) = (\{q_0, q_1, q_2, q_3, q_4\}, \{q_5, q_6\}).$$

Dopo di che vado a vedere le lettere (a,b) su ogni stato, in quale classe mi riportano.

	0					1	
0	q ₀	q ₁	q ₂	q ₃	q ₄	q ₅	q ₆
a	0	1	1	0	1	1	1
b	0	0	0	0	1	0	0

Lo stato q_0 , con la lettera 'a' mi porta in q_3 , che fa parte della classe 0 (la prima classe composta da q_0, q_1, q_2, q_3 e q_4). Quindi scrivo 0.

Ora vado a vedere in ognuna delle singole classi le colonnine corrispondenti ad ogni stato. (Es.: q_0 ha la colonnina 0,0; q_1 1,0; ecc...).

Gli stati che hanno colonnine uguali e che si trovano nella stessa classe, devono rimanere insieme: quindi q_0 e q_3 hanno entrambi colonnina 0,0 e devono restare insieme.

$$\Pi_1 = (\{q_0, q_3\}, \{q_1, q_2\}, \{q_4\}, \{q_5, q_6\}).$$

Anche se q_2 e q_5 hanno la stessa colonnina rimangono separati perché si trovano già in classi diverse.

Ora ci ritroviamo con 4 classi:

classe 0 classe 1 classe 2 classe 3

$$\Pi_1 = (\{q_0, q_3\}, \{q_1, q_2\}, \{q_4\}, \{q_5, q_6\}).$$

Ripeto lo stesso identico meccanismo di prima e mi ritroverò con 5 classi.

		0	1	2	3	4	stati finali
0	q_0	q_3	q_1	q_2	q_4	q_5	q_6
a	0	0	3	3	3	3	3
b	1	2	1	1	3	1	1

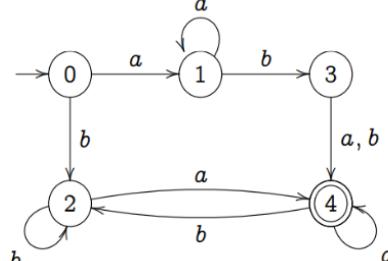
$$\Pi_2 = (\{q_0\}, \{q_3\}, \{q_1, q_2\}, \{q_4\}, \{q_5, q_6\}).$$

Rifaccio la tabellina con queste nuove classi, e mi rendo conto che non c'è più nulla da spezzare. Il partizionamento è finito. Gli stati dell'automa saranno le classi di equivalenza con indice 0, 1, 2, 3, 4.

		0	1	2	3	4
0	q_0	q_3	q_1	q_2	q_4	q_5
a	1	1	4	4	4	4
b	2	3	2	2	4	2

$$\Pi_3 = \Pi_2.$$

Il nuovo automa minimizzato è il seguente, e avrà 5 stati invece dei 7 iniziali.



Grammatiche Regolari

Una grammatica $G = \langle V, \Sigma, P, S \rangle$ è di tipo 3 se tutte le produzioni sono di una di queste forme

$X \rightarrow aY$ (variabile \rightarrow terminale seguito da variabile)

$X \rightarrow a$ (variabile \rightarrow solo terminale)

$X \rightarrow \epsilon$ (ammesse solo per $x = S$ e solo se S non compare mai a destra)

Un linguaggio è regolare se e solo se è generato da una grammatica di tipo 3. Grazie all'equivalenza con automi a stati finiti, possiamo sempre passare da una descrizione all'altra.

Dalle grammatiche all'automa

- Se $G = \langle V, \Sigma, P, S \rangle$ una grammatica di tipo 3 e per semplicità, suppongo che non abbia la produzione $S \rightarrow \epsilon$
- costruisco l'automa A non deterministico:
 - gli stati sono le variabili della grammatica G e lo parola vuota
 - per ogni produzione $X \rightarrow aY$, nel grafo di A c'è la freccia $X \xrightarrow{a} Y$
 - per ogni produzione $X \rightarrow a$, nel grafo di A c'è la freccia $X \xrightarrow{a} \epsilon$
 - lo stato iniziale è S
 - l'unico stato finale è ϵ
- si verifica che $L(G) = L(A)$

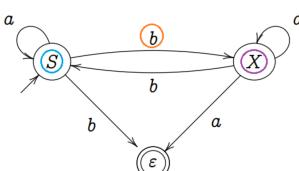
Se $S \rightarrow \epsilon$ (è una produzione di G (e quindi S non compare nei letti destri delle produzioni)), allora occorrerà aggiungere S all'insieme degli stati finali di A .

ESEMPIO: DALLA GRAMMATICA ALL'AUTOMA

Se G ha le produzioni

$S \rightarrow aS$, $S \xrightarrow{b} X$, $S \rightarrow b$, $X \rightarrow aX$, $X \rightarrow bS$, $X \rightarrow a$,

allora $L(G)$ è accettato dall'automa



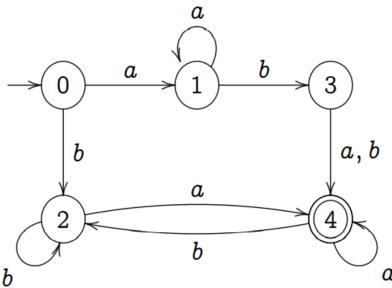
Dall'automa alla grammatica

- Se $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ un autome a stati finiti, privo di ϵ -transizioni, per semplicità $\epsilon \notin L(A)$
- costruisco la grammatica G nel seguente modo:
 - le variabili della grammatica G sono gli stati dell'autome A
 - per ogni freccia $x \xrightarrow{a} y$ nel grafo di A aggiungiamo a G la produzione $x \rightarrow a y$
 - se y è uno stato finale, aggiungiamo a G anche la produzione $y \rightarrow \epsilon$
 - il simbolo iniziale è lo stato iniziale $S = q_0$
- Si verifica che $L(G) = L(A)$ (dimostrazione analogia al caso preced.)

Se $\epsilon \in L(A)$, devo modificare la grammatica G in modo da fargli generare anche le parole vuote. Se però S compare nel lato destro di qualche produzione, non è possibile aggiungere la produzione $S \rightarrow \epsilon$. Introduco quindi la nuova variabile S' le cui produzioni avranno gli stessi letti destri delle produzioni di S . Prendo quindi S' come simbolo iniziale al posto di S e aggiungo la produzione $S' \rightarrow \epsilon$.

ESEMPIO: DALL'AUTOMA ALLA GRAMMATICA

Dall'automa



si ottiene la grammatica con simbolo iniziale X_0 e produzioni:

$$\begin{array}{lllll} X_0 \rightarrow aX_1 & X_1 \rightarrow bX_3 & X_2 \rightarrow bX_2 & X_3 \rightarrow b & X_4 \rightarrow a \\ X_0 \rightarrow bX_2 & X_2 \rightarrow a & X_3 \rightarrow a & X_3 \rightarrow bX_4 & X_4 \rightarrow aX_4 \\ X_1 \rightarrow aX_1 & X_2 \rightarrow aX_4 & X_3 \rightarrow aX_4 & X_4 \rightarrow bX_2 & \end{array}$$

Il linguaggio generato da tale grammatica coincide con quello accettato dall'automa.

Grammatiche Lineari

Una grammatica si dice lineare se tutte le produzioni hanno la forma:

$$X \rightarrow uYv \quad o \quad X \rightarrow u, \text{ con } X, Y \in N \text{ e } u, v \in \Sigma^*$$

Una grammatica si dice lineare destro se tutte le produzioni hanno la forma:

$$X \rightarrow uY \quad o \quad X \rightarrow u, \text{ con } X, Y \in N \text{ e } u \in \Sigma^*$$

In altre parole, una grammatica è lineare se le produzioni hanno a sinistra una variabile, e a destra hanno al più una variabile; è lineare destro se tali variabili compiono solo alla fine di tale termine (alla fine delle parole).

Le grammatiche di tipo 3 considerate finora, sono evidentemente lineari destri. Viceversa si può dimostrare che ogni grammatica lineare destro, genera un linguaggio di tipo 3.

Lemme di iterazione

d) Una proprietà dei linguaggi regolari è il Lemme di iterazione

Se prendiamo un linguaggio regolare, possiamo trovare un intero m per cui tutte le parole del linguaggio che sono più lunghe di questo intero m , possono essere scomposte in 3 pezzi: xyz .

Dove la parte centrale y non è la parola vuota, ma un fattore che può essere ripetuto m volte senza che si esca dal linguaggio.

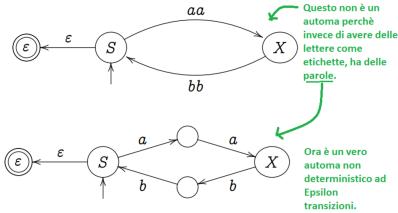
Un linguaggio che non supporta queste proprietà di iterazione (pumping), non può essere regolare, anche se esistono linguaggi che hanno le suddette proprietà ma non sono regolari.

ESEMPIO:

Sia G la grammatica con le produzioni

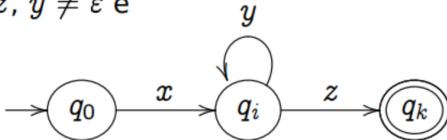
$$S \rightarrow aaX, \quad S \rightarrow \epsilon, \quad X \rightarrow bbS.$$

L'automa che accetta $L(G)$ si ottiene nel modo seguente:



Dimostrazione

- Per il Teorema di Kleene, L è accettato da un automa a stati finiti \mathcal{A} ,
- sia n il numero degli stati,
- sia $w \in L$ e $|w| \geq n$. Scriviamo $w = a_1 a_2 \cdots a_k$, con $a_1, a_2, \dots, a_k \in A$, $k \geq n$, $k = \text{numero delle lettere}$ prendiamo una parola che abbia lunghezza $\geq n$
- nel grafo di \mathcal{A} c'è un cammino $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} q_k \in F$ c'è un cammino da uno stato iniziale ad uno finale che ha w come etichetta.
- dato che $k \geq n$, troveremo uno stato ripetuto $q_i = q_j$, $0 \leq i < j \leq k$,
- quindi, posto $x = a_1 \cdots a_i$, $y = a_{i+1} \cdots a_j$, $z = a_{j+1} \cdots a_k$ avremo $w = xyz$, $y \neq \epsilon$ e



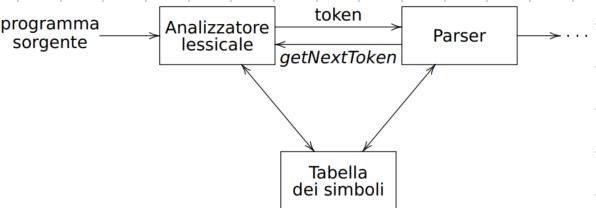
Dato che $k \geq n$, ci saranno almeno $n+1$ stati. Quindi questi stati sono più del numero di stati totale dell'automa, ci saranno di conseguenza delle ripetizioni.

- pertanto \mathcal{A} accetta tutte le parole $xy^n z$ con $n \geq 0$,
- cioè $xy^n z \in L$ per ogni $n \geq 0$.

Analisi Lessicale

Come visto inizialmente, l'analisi è la prima fase nella compilazione di un codice serpente. Bisogna dunque identificare gli elementi atomici che costituiscono il mostro codice e individuare a quale categorie lessicale fanno riferimento.

Il programma serpente viene dato in input all'analizzatore lessicale che restituisce un token al parser ogni volta che questo deve procedere con le sue analisi sintattiche. Contemporaneamente all'invio del token, l'analizzatore scrive nella tabella dei simboli gli attributi del token, che poi potranno essere recuperati dal parser.



6. È importante separare l'analisi lessicale da quelle sintattica perché si ha un'enorme semplificazione nelle progettazioni del compilatore, una maggiore efficienza e portabilità rispetto a che se fossero svolte insieme

TOKEN: simbolo estratto che rappresenta un'unità lessicale (ad esempio una parola chiave) I simboli processati dal parser sono i tokem. Ad ogni tokem viene assegnato un pattern

LESSEMA: è una sequenza di caratteri di un codice sorgente associata a un tokem. L'analizzatore lessicale identifica i lesseni come istanze dei tokem a cui sono associati.

PATTERN: è una descrizione delle forme che i lesseni possono avere per poter essere associati ad un determinato tokem, in pratica si tratta di un'espressione regolare

NB Se più di un lesseno è associato al medesimo pattern, è necessario tenere traccia di ulteriori informazioni sul particolare lesseno letto dall'analizzatore lessicale. Queste info sono salvate nelle **Tabelle dei simboli**



token	descrizione informale	esempi
if	if	if
else	else	else
comparazione	<, >, <=, >=, == e !=	<, >, <=, ...
id	lettera seguita da lettere e cifre	valore, a, c1, ...
numero	costanti numeriche	3.14159, 0, 6.02e23
stringa letterale	qualunque cosa tra due "	"hello", "Toni", ...

Esempio: consideriamo l'espressione $E = M * C^{**} 2$. Questa produrrà la seguente serie di token e attributi.

1. < id , puntatore alla riga di E nella tabella dei simboli >
2. < op assegnazione >
3. < id , puntatore alla riga di M nella tabella dei simboli >
4. < op prodotto >
5. < id , puntatore alla riga di C nella tabella dei simboli >
6. < op potenza >
7. < numero , valore intero 2 >

Analisi lessicale : 2 compiti principali

→ Spezzare il testo in token, cioè unità sintattiche fondamentali (parole chiave, identificatori, numeri...)

→ Per ogni parte del testo (chiamate lessenne), trovare il token corrispondente secondo:

- massima lunghezza del match possibile
- le priorità più alte tra i pattern che matchano

Regole

Quando l'analizzatore esamina il testo:

- cerca il più lungo lessenne iniziale che corrisponde a un pattern
- fra i token compatibili con quel lessenne, sceglie quello con la priorità maggiore
- restituisce quel token e passa al resto del testo, ripetendo il processo

int interesse = 5; → token ottenuti: int, identificatore, =, numero, ;.

1) leggo da sinistra, int è lessenne più lungo iniziale che matcha un pattern (potrebbe essere sia uno keyword int che un identificatore)

2) int parola chiave → maggiore priorità rispetto identificatore

3) Restituito token int

:

L'analizzatore lessicale non si limita a trovare un match

- cerca sempre il lessenne più lungo possibile
- restituisce il token con priorità maggiore fra quelli compatibili.

Linguaggi Non Contestuali

Sono linguaggi di tipo 2, e lo strumento necessario che ci permette l'analisi sintattica.

Una grammatica si dice non contestuale o di tipo 2 (CFG: Context Free Grammar), se tutte le produzioni sono delle forme:
 $X \rightarrow Q$, con $X \in N$, $Q \in V^*$

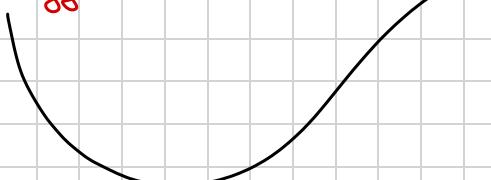
→ hanno a sinistra una singola variabile

Un linguaggio è non contestuale o di tipo 2 (CFL) se è generato da una grammatica non contestuale.

Ogni grammatica lineare destra, e in generale le grammatiche lineari, sono non contestuali, perché a sinistra delle produzioni c'è sempre una variabile sola.

Quindi, dato che sono generati da grammatiche regolari dx, i linguaggi regolari sono non contestuali e costituiscono un sottoinsieme dei linguaggi non contestuali. No viceversa.

Tra i veri linguaggi non contestuali ce n'è uno che rappresenta l'archetipo di questo tipo di linguaggi: il **linguaggio di Dick**.



→ si usi per raggruppare le produzioni con lo stesso letto sinistro, separandole i letti destri con una barre.
es: $S \rightarrow a|b|θ|(S+S)|SS|S^*$
per dire S produce a, b, θ ...

Linguaggio di Dick: $\Sigma_m = \{a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_m\}$.

Sia $G_m = \langle V, \Sigma_m, P, S \rangle$ la grammatica con unica variabile S e produzioni:

$S \rightarrow a_i S b_i \quad , \quad i = 1, 2, \dots, m$

$S \rightarrow SS$

$S \rightarrow ε$

Il linguaggio D_m generato da G_m è detto linguaggio di Dick.

Alberi di derivazione

Rappresentazione di una derivazione di una parola in una grammatica non contestuale.

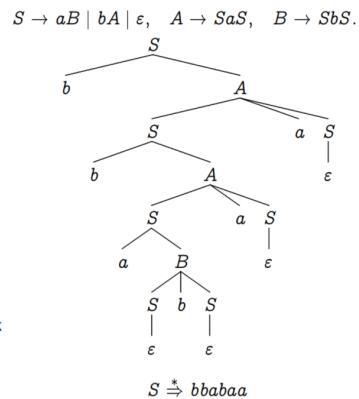
Un albero è un insieme di modi, tra i quali ce n'è uno chiamato radice, e poi c'è una relazione tra i modi per cui a ogni modo è associata una sequenza di figli. Ogni modo può avere tanti figli, ma ogni figlio ha un solo genitore; la radice non ha genitore. I figli di ciascun modo hanno un ordinamento, e i modi privi di figli sono detti foglie, anche queste ordinate. Tutti i modi hanno delle etichette, e la radice ha come etichetta il simbolo iniziale della grammatica.

Definizione

T è un albero di derivazione della grammatica G se verifica le seguenti condizioni:

- 1 l'etichetta della radice è S ,
(variabili)
- 2 le etichette dei nodi interni sono elementi di N ,
- 3 le etichette delle foglie sono elementi di $\Sigma \cup \{\epsilon\}$,
- 4 le foglie con etichetta ϵ sono 'figli unici',
simboli terminali o parola vuota. Nel secondo caso la foglia non deve avere fratelli e deve essere l'unico figlio del suo genitore.
- 5 se un nodo con etichetta X ha figli etichettati nell'ordine $\alpha_1, \dots, \alpha_k$, allora $X \rightarrow \alpha_1 \dots \alpha_k$ è una produzione di G .
nella grammatica deve esserci la produzione X produce alfa1, ..., alfak

La parola che si ottiene leggendo, nell'ordine, le etichette delle foglie è la parola associata a T .



Una parola appartiene ad un linguaggio generato dalla grammatica se e solo se esiste un albero di derivazione della grammatica, associato alle suddette parole.

Una grammatica si dice non ambigua se ad ogni parola del linguaggio generato corrisponde un unico albero di derivazione.

Al contrario, le grammatiche ambigue sono quelle grammatiche dove ad ogni parola del linguaggio generato, corrispondono più alberi di derivazione.

Semplificazione di grammatiche non contestuali

Quando si ha e che fare con le grammatiche, i 2 problemi principali sono quello della **recognizione** (stabilire se una parola fa parte del linguaggio generato) e del **parisimo** (date una parola che è nel linguaggio, trovare una derivozione).

La Complessità sta nel fatto che ci sono produzioni fastidiose:

- $X \rightarrow E$ (E -produzioni)
- $x \rightarrow Y$ (variabile produce variabile, produzioni 1-arie)

Quando si hanno produzioni così, si può cercare di creare delle grammatiche equivalenti che non hanno delle produzioni fastidiose.

Anche le variabili che non compaiono nelle forme sentenziali, e da cui non si derivano parole prive di variabili sono fastidiose.

(variabili che non producono nulla, non producono forme sentenziali)

E -produzioni e produzioni unarie

Sia G una grammatica non contestuale, le produzioni delle forme:

- $X \rightarrow E$ (E -produzioni)
- $x \rightarrow Y$ (variabile produce variabile produzioni unarie)

L'ideale sarebbe avere a sx sempre una variabile e a dx avere un termine, o una sequenza di almeno due lettere variabile-termine.

In sostanza di produzioni unarie e E -produzioni, se $\alpha \Rightarrow \beta$ (α primo termine, β secondo termine) allora:

- $|\beta| > |\alpha|$ oppure
- $|\beta| = |\alpha|$ ma β contiene un termine in più di α

Una parola di lunghezza n può essere generata in $2n-1$ passi al più (ad ogni passo la mia forma sentenziale si allunga, oppure una variabile viene sostituita da un termine).

Ogni linguaggio non contestuale è possibile generarlo da una grammatica non contestuale priva di ϵ -produzioni, tranne, la produzione $S \rightarrow \epsilon$ (se presente), ove S è il simbolo iniziale.

Inoltre si può assumere che il simbolo iniziale non compaia nei cati destri delle produzioni.

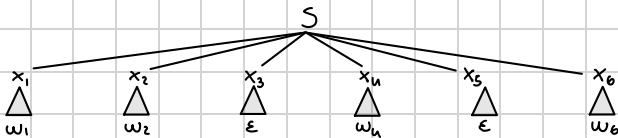
Le produzioni $S \rightarrow \epsilon$ servono solo per generare la parola vuota, e può essere usata solo come primo passo, se S non compare nei cati destri di nessuna produzione. Altrimenti non le si può usare, è come se non esistesse.

Eliminazione delle ϵ produzioni

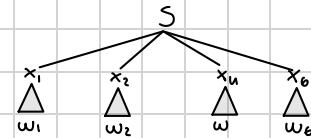
Una variabile si dice omnullabile se genera una la parola vuota. Questo vale non solo per le produzioni dirette come $x \rightarrow \epsilon$, ma anche per le produzioni concatenate: $x \rightarrow y ; y \rightarrow w ; w \rightarrow \epsilon$.

Quando costruisco una grammatica, posso avere diversi casi:

- i) S non è omnullabile, quindi non è possibile generare la parola vuota con il linguaggio. Perendo di una grammatica G non semplificata, costruiamo una nuova grammatica G' in questo modo:
 - aggiungiamo a G' tutte le produzioni che si ottengono cancellando nei lati dx delle produzioni di G , una o più occorrenze di variabili omnullabili.
 - cancelliamo le ϵ produzioni, in questo modo G' sarà identico a G anche se ho tolto tutte le ϵ -produzioni.



$S \rightarrow x_1 x_2 x_3 x_4 x_5 x_6$ in G e x_3 e x_5 sono omnullabili



In G' ho le produzioni:
 $S \rightarrow x_1 x_2 x_4 x_6$

2) S è omnullabile. A partire da G costruiamo una nuova grammatica G' nel seguente modo:

- procedendo come nel caso precedente si ottiene una grammatica G' che genera le stesse parole di G , eccetto ϵ ;
- se S non compone nei letti destri, basta aggiungere la produzione $S \rightarrow \epsilon$
- Se S compone nei letti destri, si aggiunge una nuova variabile S' , che sarà il nuovo simbolo iniziale e le produzioni $S' \rightarrow \epsilon$ e $S' \rightarrow S$. In questo modo ho fatto scomparire il simbolo iniziale dei letti dx delle produzioni, e posso aggiungere ϵ .

Ricerca delle variabili omnullabili

ESEMPIO

Sia G la grammatica con $N = \{S, O, P, E\}$, $\Sigma = \{a, b, x\}$ e con le produzioni

$$S \rightarrow aOb, \quad O \rightarrow P \quad | \quad aOb \quad | \quad OO, \quad P \rightarrow x \quad | \quad E, \quad E \rightarrow \epsilon.$$

O ha a dx una variabile annullabile.

P ha a dx una variabile annullabile.

derivazione diretta di epsilon.

n	W_n	$N - W_n$
1	<u>E</u>	S, O, P
2	<u>E, P</u>	S, O
3	<u>E, P, O</u>	S
4	<u>E, P, O</u>	S

Le variabili annullabili sono: E, P, O .

Quindi G è equivalente alla grammatica con produzioni

$$S \rightarrow aOb \quad | \quad ab, \quad O \rightarrow P \quad | \quad aOb \quad | \quad OO \quad | \quad ab \quad | \quad O, \quad P \rightarrow x \quad | \quad E.$$

Eliminazione delle produzioni umarie

- A partire da G costruisco una nuova grammatica G' in questo modo:
- 1) Si eliminano le ϵ -produzioni con la costruzione precedente.
 - 2) Per ogni coppia di variabili A, B tali che $A \Rightarrow^* B$ (A deriva B in uno o più passi), si aggiungono i letti destri delle produzioni di A (tutte le cose che produce B) e aggiungo a quelle che produce A)
 - 3) Si cancellano le produzioni umarie

$\rightarrow G'$ è equivalente a G perché non ho aggiunto nulla di nuovo e posso generare ciò che generava con G

Ricerca delle derivazioni umarie: per trovare tutte le derivazioni umarie (cioè del tipo $x \rightarrow^* y$ con una sola variabile a destra), costruisce un insieme di coppie (x, y) , partendo dalle produzioni dirette.

Poi aggiungo tutte le coppie transitivamente: se ho $(x, z), (z, y)$, allora aggiungo anche (x, y) . Ripeto finché non posso aggiungere più nulla. Alla fine avrò tutte le derivazioni umarie possibili tra variabili, in modo efficiente.

inaccessibili

Variabili improduttive:

- una variabile è produttiva se può generare una parola composta da soli terminali
- se non può farlo, è improduttiva \rightarrow inutile, la elimino con tutte le sue produzioni

metodo:

- 1) porto da variabili che generano solo terminali (anche ϵ)
- 2) aggiungi le variabili che producono solo terminali o altre produttive
- 3) ripeti finché l'insieme non cambia

Variabili inaccessibili:

- una variabile è accessibile se può essere raggiunta a partire da s.
- se non può essere raggiunta si dice inaccessibile \rightarrow la elimino

Procedure di riduzione:

- 1) Determinare le variabili produttive
- 2) Eliminare le variabili improduttive e le produzioni che contengono tali variabili (potrebbe generare nuove variabili inaccessibili)
- 3) Determinare le variabili accessibili delle grammatiche ottenute.
- 4) Eliminare le variabili inaccessibili e le produzioni che contengono tali variabili.

ESEMPIO

Sia G la grammatica con $N = \{E, F, T, R\}$, $\Sigma = \{+, *, -, a, (,)\}$, simbolo iniziale E e produzioni

$$E \rightarrow E + E \mid T \mid F,$$

$$F \rightarrow E * E \mid (T) \mid a,$$

$$T \rightarrow E - T,$$

$$R \rightarrow E - F.$$

Variabili produttive

	Produttive	Improduttive
n	W_n	$N - W_n$
1	F	E, T, R
2	F, E	T, R
3	F, E, R	T
4	F, E, R	T

Perchè deriva il simbolo terminale

'a'.
T non è presente perchè è già stata eliminata dato che è improduttiva.

Variabili accessibili

	Accessibili	Inaccessibili
n	K_n	$N - K_n$
0	E	F, R
1	F, E	R
2	F, E	R

$$E \rightarrow E + E \mid F,$$

$$F \rightarrow E * E \mid a.$$

$$R \rightarrow E - F.$$

Forme Normali di Chomsky

Una grammatica non contestuale si dice in forme normali di Chomsky se ha solo produzioni del tipo:

- $X \rightarrow YZ$ con $X, Y, Z \in N$, quindi abbiamo una variabile e sx e una coppia di variabili e dx
- $X \rightarrow \alpha$, con $X \in N$, $\alpha \in \Sigma$, abbiamo una variabile e sx e un simbolo terminale e dx
- $S \rightarrow \varepsilon$, ma solo è condizione che S non compare nei letti destri delle produzioni.

Esempio: $S \rightarrow AB|\varepsilon$, $A \rightarrow AA|\alpha$, $B \rightarrow BB|\beta$

Negli alberi di derivazione di una grammatica in forme normali di Chomsky le foglie sono figli unici, e i nodi interni costituiscono un albero binario completo, perché ogni variabile che non ha il simbolo terminale avrà due variabili

Ogni linguaggio non contestuale è effettivamente generato da una grammatica non contestuale in forme normali di Chomsky.

Le procedure funziona così:

- 1) Eliminare le ε produzioni e produzioni unarie; restano solo produzioni "variabile produce terminale" $X \rightarrow \alpha$ con $X \in N$ e $\alpha \in \Sigma$ e produzioni "variabile produce letto dx che contiene almeno due lettere terminale-variabile" $X \rightarrow \gamma$ con $|\gamma| \geq 2$
- 2) Ridursi al caso $\gamma \in N^*$ (γ ha solo variabili). Per ogni terminale α che compare, non ha solo in qualche letto dx:
 - introduco una nuova variabile A e una nuova produzione $A \rightarrow \alpha$
 - sostituisco tutte le occorrenze di α nei letti dx delle produzioni con A.
- 3) Voglio che le variabili e dx siano solo 2. Introduco quindi una nuova variabile e sostituisco le produzioni con le coppie di produzioni

ESEMPIO

Sia G la grammatica con produzioni

$$S \rightarrow aOb \mid ab, \quad O \rightarrow \underline{aOb} \mid \underline{OO} \mid \underline{ab} \mid x.$$

Con la procedura indicata, otteniamo

$$S \rightarrow AOB \mid AB, \quad O \rightarrow AOB \mid OO \mid AB \mid x, \quad A \rightarrow a, \quad B \rightarrow b.$$

In questo modo riesco a creare una grammatica che nei lati ha solo ed almeno due variabili.

creo due nuove variabili

Sia G la grammatica con produzioni

$$S \rightarrow AOB \mid AB, \quad O \rightarrow AOB \mid OO \mid AB \mid x, \quad A \rightarrow a, \quad B \rightarrow b.$$

Con la procedura indicata, otteniamo

$$S \rightarrow A\textcircled{Z} \mid AB, \quad O \rightarrow A\textcircled{Z} \mid OO \mid AB \mid x, \quad \textcircled{Z} \rightarrow OB, \quad A \rightarrow a, \quad B \rightarrow b.$$

La grammatica ottenuta, equivalente a G , è in forma normale di Chomsky.

Forma Normale di Greibach: una grammatica si dice in forma normale di Greibach se ha solo produzioni dei tipi.

- $X \rightarrow \alpha f$, con $\alpha \in \Sigma$ e $f \in N^*$
- $S \rightarrow \varepsilon$, ma solo a condizione che S non compaia nei lati destri delle produzioni.

→ $S \rightarrow aSB \mid aB$, $B \rightarrow \varepsilon$ in forma normale di Greibach

Lemme di iterazione per i Linguaggi Non contestuali

Il lemma per i linguaggi regolari consistevo in:

Sia L un linguaggio regolare, esiste un intero m tale che ogni parola $w \in L$ di lunghezza $|w| \geq m$, si può fattorizzare $w = xyz$, con $y \neq \varepsilon$ e $xy^mz \in L \quad \forall m \geq 0$

Il linguaggio $L = \{a^m b^m \mid m > 0\}$ non è regolare in quanto non mi posso ricondurre alle forme $xyz \in L$.

Il lemma di iterazione per i linguaggi non contestuali:

sia L un linguaggio non contestuale. Esiste un intero m tale che ogni parola $w \in L$ di lunghezza $|w| \geq m$ si può fattorizzare $w = xuyvz$ con $u, v \neq \varepsilon$ e $xu^k y v^k z \in L \quad \forall k \geq 0$.

In questo caso itero parallelamente u e v (pumped)

Dimostrazione

Mostrirete che se una parola è sufficientemente lunga, allora può essere "pompata" (cioè si possono iterare certi pezzi) senza uscire dal linguaggio

1) Fisso un limite m

- Sia m le lunghezze massime delle porote di L che hanno un albero di derivazione di altezza $\leq \ln L$, cioè massimo tante variabili quante ne ha la grammatica.
- Questo limite m serve per scegliere porote sufficientemente grandi da forzare le ripetizioni di una variabile.

2) Prendo una parola $w \in L$ con $|w| > m$.

Questo garantisce che, in ogni commimo nell'albero di derivazione, una variabile deve comparire almeno 2 volte

3) Ci sono una variabile che si ripete: chiamiamola A .

Nella derivazione avremo:

$$S \Rightarrow * x A z$$

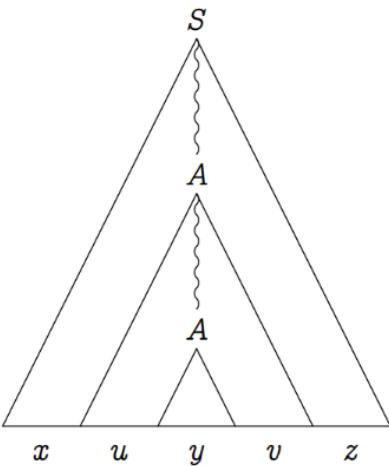
$$A \Rightarrow * u A v$$

$$A \Rightarrow * y$$

Quindi posso scrivere la parola come: $w = x u y v z$ con $u, v \neq \epsilon$

4) Iteriamo la parte centrale

Ripetendo la parte centrale $u A v$, ottengo: $u^k u^k y v^k v^k z \in L \forall k \geq 0$



L'albero mostra le ripetizioni della variabile A lungo un commimo discendente:

- Il sottoalbero di $A \Rightarrow * u A v$ si ripete
- Ogni iterazione aggiunge un u , uno y e un v

Algoritmo di Cocke-Kasami-Younger (CYK)

Serve per verificare se:

- una parola $w \in L(G)$ ha riconoscimento
- costruire le parsing (cioè un albero di derivazione)
- funzione su grammatiche in forma normale di Chomsky

↳ Date una grammatica G in FNC (produzioni solo del tipo $A \rightarrow BC$, $A \rightarrow e$) e una parola $w = a_1 a_2 a_3 \dots a_n$.

Costruisco una tabella triangolare N_{ij} , dove:

N_{ij} è l'insieme delle variabili che generano il fattore $a_{i+1} \dots a_j$.

Algoritmo:

caso base $\rightsquigarrow A \in P$, metti in N_{ii} tutte le variabili X per cui $X \rightarrow a_{i+1} \in P$.

passo induttivo (ricorsione) $\rightsquigarrow A$ intervallo più lungo (di lunghezza ≥ 2), calcola:

$$N_{ij} = \bigcup_{h=i+1}^{j-1} N_{ih} \cdot N_{hj}$$

Dove l'operazione \circ corrisponde a: $YZ = \{x \in N \mid XYZ\}$, è una specie di operazione di concretizzazione.

Verifica \rightsquigarrow Se $s \in N_{0m}$, allora le parole appartiene al linguaggio: $w \in L(G)$

esempio. Verifico se la parola $w = aabb \in L(G)$ con CYK

$$S \rightarrow XY \mid AY \mid XB \mid AB \mid AS \quad | \quad a, a, a, a = aabb$$

$$X \rightarrow e$$

$$Y \rightarrow BS$$

$$A \rightarrow e$$

$$B \rightarrow b$$

per ogni lettera a_i , guardo quali variabili le producono direttamente

- 4) prendo le lettere singole e controllo da cosa sono prodotte
 $a \rightsquigarrow A$; $b \rightsquigarrow B$
- 2) Aumento le dimensioni dei gruppi e prendo tutte le possibili combinazioni.
 $aa, \alpha, \alpha \rightsquigarrow AA$. Non ci sono produzioni che mi generano AA; proseguo con $ab, \alpha, b \rightsquigarrow AB$ che è prodotto da s ...
- 3) Aumento ancora e prendo tutti i sottogruppi, controllando sempre da cosa sono prodotti. Avremo così fino alla fine.
- 4) Controllo sull'ultimo blocco se è presente s, in tal caso significa che la parola w è accettata dalla grammatica.

4	S			
3	S	O		
2	O	S	O	
1	A	A	B	B
w =	a	a	b	b

Parsing Top-Down

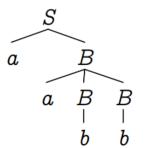
In genere l'operazione di Parsing richiede un tempo notevole $O(n^3)$ e non è accettabile nel corso d'essere fatto fare la compilazione di un programma molto complesso (perché come sappiamo, raramente un programma non presenta errori alla prima compilazione, quindi dovremmo fare più tentativi oppure effettuare la compilazione e il parsing più volte).

Il **parsing-top-down** è una tecnica per verificare se una parola appartiene a un linguaggio (generato da una grammatica) cercando di costruire una derivazione della radice (S) verso la parola.

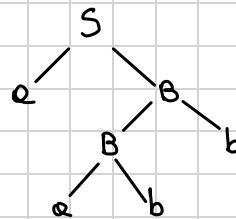
Input: una grammatica $G = \langle V, \Sigma, P, S \rangle$ e una parola $w \in \Sigma^*$
Output: una derivazione sinistra $S \Rightarrow a_1 \Rightarrow \dots \Rightarrow w$

Esempio

$S \rightarrow aB \mid bA, \quad A \rightarrow a \mid aS \mid bAA, \quad B \rightarrow b \mid bS \mid aBB.$



$S \Rightarrow aB \Rightarrow aaBB \Rightarrow \underline{aabB} \Rightarrow aabb,$
 $S \Rightarrow aB \Rightarrow aaBB \Rightarrow \underline{aaBb} \Rightarrow aabb,$



NB: anche se ci sono più derivazioni, se usiamo solo derivazioni sinistre, otteniamo una sola derivazione \Rightarrow un solo albero.
Questo è utile per rendere univoca la costruzione dell'albero, quindi è preferito nel parsing.

Endmarker - marcatori di fine parola

Quando si fa il parsing, ci serve un modo per sapere quando la parola è terminata. Per farlo aggiungo:

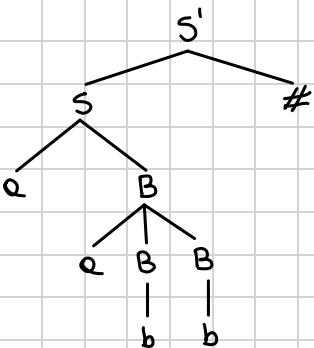
- un nuovo simbolo iniziale S'
- un nuovo termine $\#$
- una produzione: $S' \rightarrow S\#$

Così invece di derivare $aabb$, deriviamo $aabb\#$ e quando raggiungo il $\#$ so che ho finito

esempio di parsing top-down di aabb

$$\begin{array}{l} S' \rightarrow S\# \\ S \rightarrow aB1bA \\ A \rightarrow a1eS1bAA \\ B \rightarrow b1bS1eBB \end{array} \left. \begin{array}{l} \text{parsing} \rightarrow S' \Rightarrow S\# \text{ (inizio)} \\ \Rightarrow aB\# \text{ (S} \rightarrow aB) \\ \Rightarrow aaBB\# \text{ (B} \rightarrow aBB) \\ \Rightarrow aabB \text{ (B} \rightarrow b) \\ \Rightarrow aabb \text{ (B} \rightarrow b) \end{array} \right\}$$

parole accettata, i.e. $\#$ è stato derivato correttamente



Si chiama parsing top-down proprio perché siamo partiti dalla radice per costituire l'albero. Fossimo partiti dalle foglie, sarebbe stato un parsing bottom-up

Parsing Predittivo

Il parsing predittivo è una tecnica top-down per analizzare se una parola appartiene a un linguaggio generato da una grammatica.

idea di base:

- Abbiamo una predizione (la parte della derivazione che va ancora completata).
- Abbiamo l'input da analizzare
- A ogni passo guardiamo il simbolo più a sinistra della predizione:
 - se è una variabile, lo sostituisco con una delle produzioni
 - se è un terminale, lo confrontiamo con la prima lettera dell'input.
 - se coincidono → consumiamo il simbolo
 - se no → fallimento (torno indietro o la grammatica no predittive)
- Ripeto fino a esaurire input e predizione.

$w = aabc\#$ $G = S' \rightarrow S\#$ $S \rightarrow AB\mid DC$ $A \rightarrow a\mid eA$ $B \rightarrow b\mid bBc$ $C \rightarrow c\mid cC$ $D \rightarrow ab\mid eDb$

Input	Analisi	Predizioni
aabc#	/	s'
aabc#	/	s#
aabc#	/	AB#
aabc#	/	eAB#
abc #	e	AB#
abc #	e	eB#
bc #	ee	B#
bc #	ee	bc#
c #	eeb	c#
#	eebc	#
/	eebc #	/

Le parole appartiene al linguaggio

Il parsing predittivo rispetto ad un top-down classico è più efficiente, la complessità è $O(n)$. La sua esecuzione è step-by-step con delle regole. Si tiene in considerazione il lookahead (primo simbolo non ancora analizzato).

Il top-down classico è più un concetto teorico, quello predittivo invece viene applicato realmente per costruire parser.

come costruire un parser:

~> **parser e diverse ricorsive**: consiste nello scrivere tante procedure quante sono le variabili della nostra grammatica
 es. interpreto $S \rightarrow aB\mid bA$ come : a ha successo e poi B o successo oppure b ha successo e poi A ha successo

È ideale per la programmazione di un parser per una grammatica specifica. Ha una procedura per ogni variabile.

Dobbiamo tenere traccia:

- posizione corrente nella produzione (automatico)
- input (globale)
- posizione corrente nell'input (globale)
- posizione nell'input al momento chiamata (esole)

$G = S' \rightarrow S\#$	passo	active rules	sentence	parse
$S \rightarrow DC1AB$				$S' \rightarrow S\#$
$D \rightarrow ab \mid aDb$	1	$S' \rightarrow S\#$	$abc \#$	$S' \rightarrow S\#$
$C \rightarrow c \mid cC$	2	$S \rightarrow DC$	$abc \#$	$S \rightarrow DC$
$B \rightarrow bc \mid bBc$	3	$D \rightarrow ab$	$abc \#$	$D \rightarrow ab$
$A \rightarrow a \mid cA$	u	$C \rightarrow c$	$abc \#$	$C \rightarrow c$

$w = abc \#$

Rispetto al parsing predittivo qui lo si esegue in modo ricorsivo, viene quindi implementato direttamente nel codice. Il parsing predittivo invece lavora con matrici e lookahead per le decisione delle mosse.

