Compilatori

F. M. Malvestuto

Corso di Laurea Magistrale in INFORMATICA

A. A. 20013/2014

INDICE

Cap. 1

LINGUAGGI FORMALI

Un *alfabeto* Σ è un insieme finito e nonvuoto di elementi (detti *caratteri* o segni grafici). Una *stringa* su Σ di lunghezza k, $k \geq 1$, è una sequenza di n termini appartenenti ad Σ, è cioè una disposizione con ripetizione di elementi Σ di classe k.
L'insieme delle stringhe su Σ di lunghezza k, $k \geq 1$, è indicato con Σ^k e prende il nome di *potenza k-esima* di Σ. Se n è la cardinalità di Σ, allora Σ^k contiene n^k stringhe.
Introduciamo per completezza anche una stringa di lunghezza zero, che chiamiamo stringa *vuota* ed indichiamo con ε.
Se x e y sono due stringhe rispettivamente di lunghezza m ed n, con xy indichiamo la stringa di lunghezza $m+n$ che si ottiene giustapponendo y ad x. La stringa xy è vista come il risultato di un'operazione binaria, detta *concatenazione*, che è associativa ma non commutativa. Una stringa (eventualmente vuota) y è una *sottostringa* (o un *affisso*) di una stringa x se esistono due stringhe u e v su Σ tali che $x = uyv$; in tal caso, la coppia di stringhe (u, v) definisce il *contesto* di y. Una sottostringa y di x è un *prefisso* o un *suffisso* se il contesto (u, v) di y è rispettivamente della forma (ε, v) o (u, ε). Una sottostringa che non sia né un prefisso né un suffisso prende il nome di *infisso*.
Siano

$$\Sigma^+ = \bigcup_{n = 1, \ldots, \infty} \Sigma^n \qquad\qquad \Sigma^* = \{\varepsilon\} \cup \Sigma^+.$$

Se indichiamo con il simbolo "$\cdot$" l'operazione di concatenazione tra stringhe, la coppia $(\Sigma^+, \cdot)$ è un semigruppo e la coppia $(\Sigma^*, \cdot)$ è un monoide di cui la stringa vuota è l'elemento neutro.
Proviamo ora che Σ^* è un insieme infinito numerabile. Supponiamo di ordinare in maniera arbitraria gli elementi di Σ, e ne sia $(a_1, \ldots, a_n)$, dove $n = |\Sigma|$ un ordinamento (tipicamente, l'ordinamento "alfabetico"). Possiamo allora numerare gli elementi di Σ^* alla maniera seguente

stringa	numero d'ordine
ε	0
a_1	1
a_2	2
...	
a_n	n

a_1a_1	n+1
a_1a_2	n+2
...	
a_1a_n	2n
...	

Un siffatto ordinamento di Σ^* è chiamato *ordinamento lessicografico* (relativo all'ordinamento di Σ) e stabilisce una corrispondenza biunivoca tra Σ^* e l'insieme degli interi nonnegativi. Ne segue che Σ^* è un insieme infinito numerabile.

Vediamo ora come determinare il numero d'ordine di una data stringa x (nell'ordinamento lessicografico di Σ^*) e come determinare la stringa in Σ^* il cui numero d'ordine sia un dato intero nonnegativo m. Per semplicità, consideriamo solo il caso $|\Sigma| = 2$. Sia (a_1, a_2) l'ordinamento alfabetico di Σ. Associamo ad ogni stringa x su $\Sigma = \{a_1, a_2\}$ la stringa y sull'alfabeto $\Omega = \{0, 1, 2\}$ così definita. Se x è la stringa vuota, allora $y = 0$; altrimenti, y si ottiene da x sostituendo ai caratteri a_1 e a_2 rispettivamente le cifre 1 e 2. Si osservi ora che se $y \neq 0$ allora y è la codifica binaria di un unico intero positivo m in cui le cifre 1 e 2 sono usate al posto delle cifre 0 e 1:

x	y	m
ε	0	0
a_1	1	$1 \times 2^0 = 1$
a_2	2	$2 \times 2^0 = 2$
a_1a_1	11	$1 \times 2^1 + 1 \times 2^0 = 3$
a_1a_2	12	$1 \times 2^1 + 2 \times 2^0 = 4$
...		
$a_1a_2a_1a_1$	1211	$1 \times 2^3 + 2 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 19$
...		

D'altra parte, dato un intero nonnegativo m, possiamo trovare la stringa x che ha numero d'ordine m nella maniera che segue. Se $m = 0$ allora x è la stringa vuota; altrimenti, dopo aver calcolato la codifica binaria y di m, x è la stringa che si ottiene da y sostituendo alle cifre 1 e 2 i caratteri a_1 e a_2.

Un *linguaggio* su di un alfabeto Σ è un qualsiasi insieme di stringhe su Σ. Pertanto, Σ^* è un linguaggio su Σ, che prende il nome di *linguaggio universale* su Σ, ed ogni linguaggio su Σ è un sottoinsieme di Σ^*. Indichiamo con $\mathcal{L}(\Sigma)$ l'insieme dei linguaggi su Σ, cosicché $\mathcal{L}(\Sigma)$ coincide con l'insieme delle parti di Σ^* e, quindi, è un insieme infinito nonnumerabile.

1.1 Linguaggi formali

Un *linguaggio formale* è un linguaggio L che è definito utilizzando un "formalismo" che consenta di identificare le stringhe appartenenti ad L.
Dato un linguaggio formale L su un alfabeto Σ, consideriamo il *problema del riconoscimento*:

> Data una stringa x su Σ, decidere se x appartiene o meno ad L.

Ora, se esiste un algoritmo che consente di risolvere il problema del riconoscimento qualunque sia la stringa x su Σ, si dice che il problema del riconoscimento per L è *decidibile* e che L è *riconoscibile*.

Come primo esempio di formalismo, prendiamo il modello computazionale basato sulle *macchine di Turing*. Data una macchina di Turing $\boldsymbol{M}$, una stringa x è *riconosciuta* da $\boldsymbol{M}$ se ha termine il processo di computazione che inizia con $\boldsymbol{M}$ nel suo *stato iniziale* e con il dispositivo di lettura con input l'inera stringa x; in tal caso, x è *accettata* o *rifiutata* se la computazione termina e termina rispettivamente in uno *stato di accettazione* di $\boldsymbol{M}$ oppure in uno stato che non è di accettazione. L'insieme delle stringhe accettate da $\boldsymbol{M}$ definisce il linguaggio *accettato* da $\boldsymbol{M}$. I linguaggi accettati dalle macchine di Turing formano la classe dei linguaggi *ricorsivamente enumerabili*. In generale, il problema del riconoscimento per un linguaggio ricorsivamente enumerabile non è decidibile, ma *semidecidibile* nel senso che ogni stringa appartenente al linguaggio è sempre accettata da $\boldsymbol{M}$, ma può esistere una stringa appartenente al linguaggio che non è né accettata né rifiutata da $\boldsymbol{M}$, cioè la sua computazione non ha termine. Un linguaggio ricorsivamente enumerabile L è *ricorsivo* se è riconoscibile da una macchina di Turing, cioè se esiste una macchina di Turing che accetti le stringhe appartenenti ad L e rifiuti quelle che non appartengono ad L.
Un altro esempio di formalismo è dato dai *sistemi generativi*, cioè da sistemi formali basati su assiomi e regole di inferenza. Diamo subito un esempio di linguaggio per il quale il problema del riconoscimento è semidecidibile ma non decidibile.

Esempio 1.1. Consideriamo il linguaggio L sull'alfabeto binario $\Sigma = \{0, 1\}$ definito dal seguente sistema formale.

Assioma: La stringa `1` appartiene ad L.
Prima regola d'inferenza: Se x appartiene ad L, allora x`0` appartiene ad L.
Seconda regola d'inferenza: Se x appartiene ad L, allora xx appartiene ad L.
Terza regola d'inferenza: Se x`111`y appartiene ad L, allora x`0`y appartiene ad L.
Quarta regola d'inferenza: Se x`000`y appartiene ad L, allora xy appartiene ad L.

Utilizzando le quattro regole d'inferenza, possiamo generare tutte le stringhe appartenenti ad L ponendo inizialmente in L la stringa `1`, e poi aggiungendo ad L le stringhe che si ottengono applicando (ove possibile) le quattro regole d'inferenza alle nuove stringhe aggiunte L.
Consideriamo ora il generico problema di riconoscimento: «data una stringa binaria x, appartiene x ad L?». Per rispondere al quesito, utilizziamo la seguente procedura deduttiva:

ISTRUZIONE 1. $M := \{1\}$.

ISTRUZIONE 2. Se x appartiene ad M, allora concludere che la stringa x appartiene ad L.

ISTRUZIONE 3. Applicare alle stringhe in M le quattro regole.
Sia N l'insieme delle nuove stringhe che così si ottengono.
Porre $M := N$ e tornare all'ISTRUZIONE 2.

Consideriamo ora il caso che x sia la stringa `0`. Seguendo la procedura deduttiva, abbiamo nei primi quattro passi i seguenti risultati:

PASSO 1. $M = \{1\}$. Siccome la stringa `0` non appartiene ad M, andiamo avanti.

PASSO 2. Applicando la prima e la seconda regola alla stringa `1`, otteniamo le stringhe `10` e `11`. Così abbiamo $M = \{10, 11\}$. Siccome la stringa `0` non appartiene ad M, andiamo avanti.

PASSO 3. Applicando la prima regola ad `10` e `11`, otteniamo le stringhe `100` e `110`.
Applicando la seconda regola ad `10` e `11`, otteniamo le stringhe `1010` e `1111`.
Così abbiamo $M = \{100, 110, 1010, 1111\}$. Siccome la stringa `0` non appartiene ad M, andiamo avanti.

PASSO 4. Applicando la prima e la seconda regola ad `100`, otteniamo le due stringhe: `1000` e `100100`.
Applicando la prima e la seconda regola ad `1010`, otteniamo le due stringhe: `10100` e `10101010`.
Applicando la prima e la seconda regola ad `110`, otteniamo le due stringhe: `1100` e `110110`.
Applicando la prima e la seconda regola ad `1111`, otteniamo le due stringhe: `11110` e `11111111`. Inoltre, la terza regola è applicabile a

`1111` in due modi distinti, cioè interpretando nella regola $x = \varepsilon$ ed $y =$ `1`, oppure $x =$ `1` e $y = \varepsilon$. Otteniamo così le due stringhe: `01` ed `10`.

Così abbiamo $M = \{$`1000`, `100100`, `10100`, `10101010`, `1100`, `110110`, `01`, `10`$\}$. Siccome la stringa `0` non appartiene ad M, andiamo avanti.

……

Va da sé che, se la stringa `0` appartenesse ad L, allora prima o poi (e comunque dopo un numero finito di passi) saremmo in grado di generare `0`. Se però `0` non appartenesse ad L, allora il nostro procedimento sarebbe inconcludente.

Considerazioni analoghe valgono anche per il metodo induttivo. In realtà la stringa `0` non appartiene ad L. Per provarlo, facciamo uso non di un algoritmo bensì di un argomento che è "fuori" del meccanismo inferenziale del sistema formale (o, meglio, "oltre"). L'argomento riguarda la seguente proprietà della funzione $f(x)$ che conta il numero di occorrenze del carattere `1` in una generica stringa binaria x.

(*Proprietà*) Per ogni stringa x appartenente ad L, $f(x)$ non è multiplo di 3, cioè non esiste nessun intero nonnegativo k tale che $f(x) = 3k$.

Questa proprietà può essere dimostrata per induzione sul numero di passi del procedimento generativo.

Passo base: $f($`1`$) = 1$ e, banalmente, la proprietà è vera.
Passo induttivo: Supponiamo che la proprietà sia vera per tutte le stringhe x generate al passo n.esimo; proviamo che essa è vera anche per le stringhe x' generate al passo (n+1)-esimo. Dunque, assumiamo come ipotesi che $f(x)$ non sia multiplo di 3. Dimostriamo che, allora, $f(x')$ non può essere multiplo di 3.

i) Se x' è ottenuta da x applicando la prima o la quarta regola di generazione, allora $f(x') = f(x)$ e, banalmente, l'assunto è vero.

ii) Se x' è ottenuta da x applicando la seconda regola, allora $f(x') = 2f(x)$. Dunque, i divisori di $f(x')$ sono 2 e tutti i divisori di $f(x)$. Di nuovo, l'assunto è vero.

iii) Se x' è ottenuta da x applicando la terza regola, allora $f(x') = f(x)$–3. Ora, se per assurdo $f(x')$ fosse multiplo di 3, allora esisterebbe un intero nonnegativo k tale che $f(x') = 3k$. Ne seguirebbe $f(x) = 3(k+1)$ e, quindi, che $f(x)$ sarebbe multiplo di 3, cosa che contrasta con l'ipotesi induttiva. Dunque, $f(x')$ non può essere multiplo di 3.

Provata la validità della proprietà di $f(x)$, possiamo senz'altro concludere che, siccome nella stringa `0` non appartiene ad L dal momento che $f(0) = 0$ e che 0 è un multiplo di 3 ($0 = 3k$ con $k = 0$).

Un tipico caso in cui il problema del riconoscimento diventa decidibile si presenta quando il processo generativo ha un andamento per così dire "monotono nondecrescente". Ne diamo ora un esempio.

Esempio 1.2. Consideriamo il seguente linguaggio L sull'alfabeto $\Sigma = \{+, =, 1\}$ specificato dal seguente sistema formale.

Assioma: Per ogni stringa u in $\{1\}^+$, la stringa $u+1=u1$ appartiene ad L.
Regola d'inferenza: Se la stringa $u+v=w$ appartiene ad L, dove u, v e w sono stringhe in $\{1\}^+$, allora anche la stringa $u+v1=w1$ appartiene ad L.

Si osservi innanzitutto che la stringa più corta tra quelle dichiarate dall'assioma è

```
1+1=11
```

che ha lunghezza 6. Si noti anche che la regola agisce in modo da produrre stringhe sempre più lunghe delle stringhe alle quali essa è applicata: più precisamente, se l è la lunghezza della stringa $u+v=w$ allora la lunghezza della stringa prodotta $u+v1=w1$ è uguale ad $l+2$. Ne segue che tutte le stringhe appartenenti ad L hanno lunghezza $l = 4+2n$ ($n \geq 1$), cioè hanno lunghezza pari e maggiore di 4. È chiaro allora come risolvere il problema del riconoscimento nel caso di una stringa x di lunghezza l arbitraria. Se l è dispari o minore di 6, possiamo concludere che x non appartiene ad L. Se $l = 6$, allora x appartiene ad L se e solo se x è la stringa `1+1=11`. Infine, se l è pari ed è maggiore di 6, diciamo $l = 4+2n$ ($n > 2$), per poter decidere se x appartiene o meno ad L, basterà applicare il metodo deduttivo.

ISTRUZIONE 1. $M := \{1+1=11\}$.

ISTRUZIONE 2. Per $k = 2, \ldots, n$

> Applicare la regola alle stringhe contenute in M.
> Sia N l'insieme delle stringhe che così si ottengono.
> Aggiungere ad N la stringa di lunghezza $4+2k$ della forma $u+1=u1$, con $u \in \{1\}^k$.
> Porre $M := N$.

ISTRUZIONE 3. Se x appartiene ad M allora e solo allora concludere che x appartiene ad L.

Si noti che, il numero di stringhe generate al passo k-esimo è k e quindi il numero totale di stringhe generate dall'algoritmo è

$$\sum_{k=1,\ldots,n} k = n(n+1)/2 \ .$$

Anche in questo caso possiamo seguire il metodo induttivo. Data una stringa x di lunghezza l si tenta di risalire da essa ad una delle stringhe della forma $u\texttt{+1=}u\texttt{1}$, con $u \in \{\texttt{1}\}^+$. Per quanto detto, basterà discutere il caso in cui $l = 4+2n$ ($n \geq 1$).

PROCEDURA. Confrontare x con la stringa di lunghezza l della forma $u\texttt{+1=}u\texttt{1}$ con $u \in \{\texttt{1}\}^n$. Se tale confronto dà esito positivo, possiamo concludere che la stringa x appartiene ad L; altrimenti, si prova ad interpretare la stringa x come il risultato dell'applicazione della regola ad una stringa di lunghezza l–2, cioè si prova ad esprimere x come $u\texttt{+}v\texttt{1=}uv\texttt{1}$, dove

$$u \in \{\texttt{1}\}^p,\ v \in \{\texttt{1}\}^{n-p} \qquad \text{e} \qquad 0 < p < n.$$

Se la prova dà esito negativo, allora possiamo subito concludere che x non appartiene ad L; altrimenti, ripetiamo l'intera procedura per la stringa $u\texttt{+}v\texttt{=}uv$.
Si osservi che nel caso peggiore (cioè quando la stringa x non appartiene ad L) si dovrà ripetere la procedura un numero di volte limitato da l. ■

Per finire, dato un certo formalismo, indichiamo con $\mathcal{F}(\Sigma)$ la classe dei linguaggi su Σ che possono esere definiti con quel formalismo. È chiaro che la cardinalità di $\mathcal{F}(\Sigma)$ fornisce della misura del *potere espressivo* di quel formalismo. Ovviamente, $\mathcal{F}(\Sigma)$ è un sottoinsieme di $\mathcal{L}(\Sigma)$. Qual è la cardinalità di $\mathcal{F}(\Sigma)$? Come abbiamo visto, la definizione di un linguaggio in $\mathcal{F}(\Sigma)$ può essere vista come una stringa su di un opportuno alfabeto Ω, che include l'alfabeto Σ. Ne segue che, siccome l'insieme delle stringhe su Ω è numerabile, $\mathcal{F}(\Sigma)$ è un insieme infinito numerabile, laddove $\mathcal{L}(\Sigma)$ è un insieme infinito nonnumerabile. Quindi, qualunque sia il formalismo che si adotti (cioè qualunque sia l'alfabeto Ω che si voglia adottare), il suo potere espressivo è sempre limitato ad un sottoinsieme numerabile di $\mathcal{L}(\Sigma)$.

1.2 Grammatiche

Spesso, la specifica di un linguaggio formale (ad es., di un linguaggio di programmazione) avviene utilizzando il formalismo delle grammatiche (generative) che andiamo ad introdurre.
Una *grammatica* è definita da una quadrupla $\boldsymbol{G} = (N, T, P, S)$, dove

N e T sono due insiemi finiti, nonvuoti e disgiunti,

P è un insieme finito oggetti del tipo $\lambda \rightarrow \mu$ (leggasi "λ può essere sostituito con μ"), dove λ e μ sono stringhe su $N \cup T$.
S è uno speciale elemento di *N*, chiamato *simbolo iniziale* di ***G***.

Gli elementi di *N* sono chiamati i *simboli nonterminali* (o *variabili*) di ***G***, gli elementi di *T* i *simboli terminali* di ***G***, gli elementi di *P* le *produzioni* (o "regole di sostituzione" o "regole di riscrittura") di ***G***, e il simbolo nonterminale *S* è chiamato il *simbolo iniziale* o *simbolo di partenza* (start symbol) di ***G***.
D'ora in avanti adottiamo la seguente convenzione notazionale:

Le lettere iniziali dell'alfabeto in minuscolo e corsivo (*a*, *b*, *c*, …) staranno ad indicare elementi di *T*.

Le lettere finali dell'alfabeto in minuscolo e corsivo (*x*, *y*, *z*, …) staranno ad indicare stringhe su *T*.

Le lettere iniziali dell'alfabeto in maiuscolo e corsivo (*A*, *B*, *C*, …) e la lettera *S* staranno ad indicare elementi di *N*.

Le lettere finali dell'alfabeto in maiuscolo e corsivo (*X*, *Y*, *Z*, …) staranno ad indicare stringhe sull'insieme di simboli *N*.

Le lettere dell'alfabeto greco in minuscolo (α, β, γ, …) staranno ad indicare stringhe sull'insieme di simboli $N \cup T$.

Una produzione $\lambda \rightarrow \mu$ di ***G*** si dice *applicabile* ad una stringa σ (su $N \cup T$) se λ è una *sottostringa* di σ, cioè se σ può essere scritta come $\sigma = \alpha\lambda\beta$; in tal caso, esistono $n \geq 1$ coppie di stringhe (α_1, β_1), …, (α_n, β_n) e, dato *i*, il risultato dell'*applicazione* della produzione $\lambda \rightarrow \mu$ alla stringa $\sigma = \alpha_i\lambda\beta_i$ è la stringa $\sigma' = \alpha_i\mu\beta_i$ (che si ottiene semplicemente sostituendo λ con μ). Per indicare una tale trasformazione di σ in σ', usiamo la notazione $\sigma \Rightarrow \tau$.
Una sequenza di stringhe $(\sigma_0, \sigma_1, \ldots, \sigma_k)$ (su $N \cup T$), $k \geq 0$, è una *derivazione* di σ_k da σ_0 se $k = 0$ oppure se $\sigma_{h-1} \Rightarrow \sigma_h$ per ogni h, $1 \leq h \leq k$. Siano σ e τ due stringhe; diciamo che τ è *derivabile* da σ ($\sigma \stackrel{*}{\Rightarrow} \tau$) se esiste una derivazione di τ da σ. Banalmente, se $\sigma \Rightarrow \tau$ allora $\sigma \stackrel{*}{\Rightarrow} \tau$. È facile vedere che la relazione di derivabilità gode della proprietà riflessiva e della proprietà transitiva.

Una stringa α su $N \cup T$ è una *formula* di ***G*** se α è derivabile dal simbolo iniziale *S* di ***G***, cioè se $S \stackrel{*}{\Rightarrow} \alpha$. Una formula di ***G*** è una *frase* di ***G*** se è una stringa su *T*. L'insieme delle frasi di ***G*** prende il nome di *linguaggio generato da* ***G*** che indichiamo con $L(\boldsymbol{G})$. Infine, due grammatiche sono *equivalenti* se generano lo stesso linguaggio.

Richiamiamo ora la classificazione delle grammatiche dovuta a Noam Chomsky.

Gerarchia di Chomsky

— Una *grammatica* di *tipo* 0 è una grammatica in cui ogni produzione è della forma $\lambda \rightarrow \mu$, dove λ contiene almeno un simbolo nonterminale.

Un risultato teorico che va tenuto a mente è che ogni linguaggio generato da una grammatica di tipo 0 è ricorsivamente enumerabile, e viceversa.

Esempio 1.3 Si consideri la grammatica $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}\}$, $T = \{\mathtt{a}, \mathtt{b}\}$ e $P = \{\mathtt{S} \rightarrow \mathtt{aSb}, \mathtt{S} \rightarrow \varepsilon\}$. Si lascia come esercizio la prova che le formule di $\boldsymbol{G}$ sono tutte e solo le stringhe o della forma $\mathtt{a}^n\mathtt{b}^n$ o della forma $\mathtt{a}^n\mathtt{S}\mathtt{b}^n$, con $n \geq 0$, cosicché le frasi di $\boldsymbol{G}$ sono tutte e solo le stringhe della forma $\mathtt{a}^n\mathtt{b}^n$ con $n \geq 0$. Una grammatica equivalente a $\boldsymbol{G}$ è $\boldsymbol{G}' = (N', T, P', \mathtt{S})$, dove $N' = \{\mathtt{S}, \mathtt{A}, \mathtt{B}\}$ e le produzioni in P' sono:

$\mathtt{S} \rightarrow \mathtt{ABS}$
$\mathtt{S} \rightarrow \varepsilon$
$\mathtt{BA} \rightarrow \mathtt{AB}$
$\mathtt{BS} \rightarrow \mathtt{b}$
$\mathtt{Bb} \rightarrow \mathtt{bb}$
$\mathtt{Ab} \rightarrow \mathtt{ab}$
$\mathtt{Aa} \rightarrow \mathtt{aa}$

■

— Una grammatica di tipo 0 è di *tipo* 1 se, in ogni produzione $\lambda \rightarrow \mu$, o μ è la stringa vuota oppure $|\lambda| \leq |\mu|$. In tal caso, per ogni produzione, esiste una forma "normale"

$$\beta A \gamma \rightarrow \beta \alpha \gamma \ .$$

Una siffatta produzione permette di sostituire, in ogni stringa σ che contenga il simbolo nonterminale A, un'occorrenza di A con la stringa α ogni qual volta quell'occorrenza di A si trovi nel *contesto* (β, γ). Per tale motivo, una grammatica di tipo 1 è detta anche *grammatica contestuale* (o *a struttura sintagmatica*).
Diremo che un linguaggio è *contestuale* (o *a struttura sintagmatica*) se può essere generato da una grammatica contestuale. Data la natura monotona non decrescente delle produzioni di una grammatica contestuale, ogni linguaggio contestuale è riconoscibile (cioè il problema del riconoscimento è decidibile), vale a dire, ogni linguaggio contestuale è ricorsivo; tuttavia, esistono linguaggi ricorsivi che non sono contestuali.

Esempio 1.4 Si consideri la grammatica contestuale $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$, $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ e le produzioni in P sono:

S → aSBC | aBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc

La stringa abc è una frase di ***G*** e la seguente ne è una derivazione da S:

S ⇒ aBC ⇒ abC ⇒ abc

In realtà, le frasi di ***G*** sono tutte della forma $a^n b^n c^n$ con $n \geq 1$. Ad esempio, per aaabbbccc abbiamo la seguente derivazione da S:

S ⇒ a<u>S</u>BC ⇒ aa<u>S</u>BCBC ⇒ aaaB<u>CB</u>CBC ⇒
⇒ aaaBBC<u>CB</u>C ⇒ aaaBB<u>CB</u>CC ⇒ aa<u>aB</u>BBCCC ⇒
⇒ aaa<u>bB</u>BCCC ⇒ aaab<u>bB</u>CCC ⇒ aaabb<u>bC</u>CC ⇒
aaabbb<u>cC</u>C ⇒ aaabbbc<u>cC</u> ⇒ aaabbbccc . ■

— Una grammatica di tipo 1 è di *tipo* 2 se ogni produzione è della forma $A \rightarrow \alpha$ cosicché essa permette di sostituire, in ogni stringa σ che contenga almeno un'occorrenza del simbolo nonterminale *A*, una qualsiasi occorrenza di *A* con la stringa α indipendentemente dalla posizione e dal contesto in cui essa si trovi. Per tale motivo, una grammatica di tipo 2 è detta anche *grammatica acontestuale* (o *indipendente dal contesto*).
Diremo che un linguaggio è *acontestuale* se può essere generato da una grammatica acontestuale. Un esempio di grammatica acontestuale è la grammatica ***G*** dell'Esempio 1.3.

Esempio 1.5 Si consideri la grammatica acontestuale ***G*** = (*N*, *T*, *P*, S), dove *N* = {S, A}, *T* = {a, b, c} e *P* contiene le produzioni

S → Sc | A
A → aAb | ε

Si lascia come esercizio la prova che le frasi di ***G*** sono tutte e solo le stringhe della forma $a^n b^n c^m$ per $n, m \geq 0$. ■

Esempio 1.6 Si consideri la grammatica acontestuale ***G*** = (*N*, *T*, *P*, S), dove *N* = {S, A}, *T* = {a, b, c} e *P* contiene le produzioni

S → aS | A
A → bAc | ε

Si lascia come esercizio la prova che le frasi di ***G*** sono tutte e solo le stringhe della forma $\mathtt{a}^n\mathtt{b}^m\mathtt{c}^m$ per $n, m \geq 0$. ■

— Una grammatica di tipo 2 è di *tipo* 3 se, in ogni produzione $A \rightarrow \alpha$, α contiene al più un'occorrenza di un simbolo nonterminale, e, se α contiene il simbolo nonterminale B, allora α è della forma $\alpha = xB$. Le grammatiche di tipo 3 sono anche dette *grammatiche regolari*.
Diremo che un linguaggio è *regolare* se può essere generato da una grammatica regolare.

Esempio 1.7 Si consideri la grammatica regolare ***G*** = (N, T, P, S), dove N = {S}, T = {0, 1} e P contiene le produzioni

$$\mathtt{S} \rightarrow \mathtt{0S} \mid \mathtt{1S} \mid \mathtt{1}$$

Si lascia come esercizio la prova che le frasi di ***G*** sono tutte e solo le stringhe binarie che rappresentano i numeri interi positivi dispari. ■

Cap. 2

GRAMMATICHE ACONTESTUALI

I linguaggi di programmazione sono per lo più linguaggi acontestuali. Pertanto, considereremo ora le frasi di una grammatica acontestuale e daremo sia una rappresentazione grafica della loro "struttura in costituenti immediati", sia un algoritmo polinomiale per il riconoscimento.

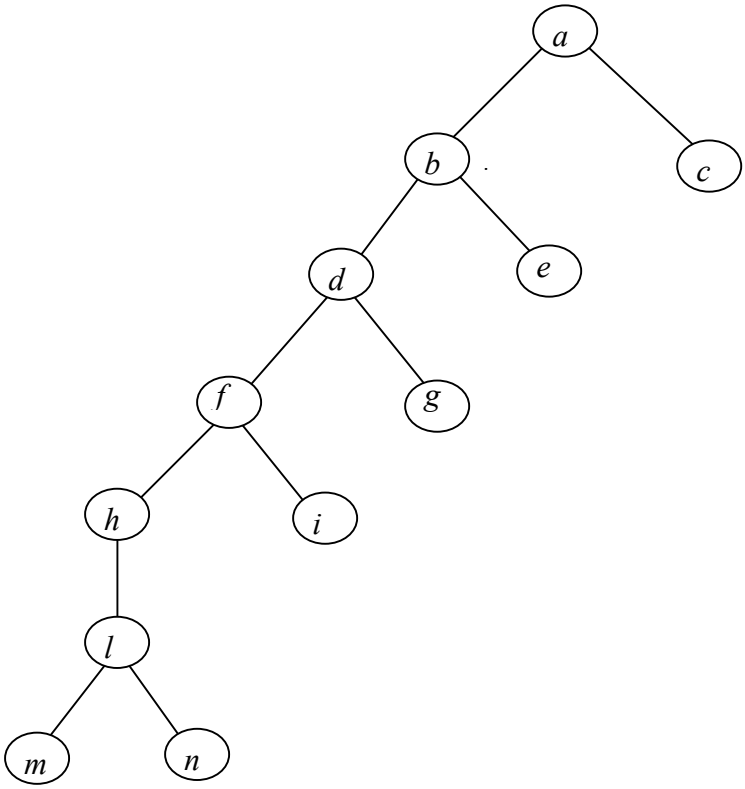
2.1 *Alberi orientati*

Un *albero* (*orientato*) è un grafo orientato T in cui:

(*i*) esiste un unico vertice, chiamato *radice* di T, che non ha archi entranti;
(*ii*) per ogni vertice v, esiste un unico cammino dalla radice di T a v.

Si osservi che la proprietà (*ii*) implica che ogni vertice distinto dalla radice ha un solo arco entrante. Chiamiamo poi *profondità* di un vertice v la lunghezza del cammino dalla radice di T a v; in particolare, la radice di T ha profondità uguale a zero.
I vertici di T che non hanno archi uscenti sono chiamati *foglie* di T. Inoltre se (u, v) è un arco di T, allora il vertice u è chiamato il *padre* del vertice v e questo, a sua volta, è detto essere un *figlio* di u. Più in generale, se esiste un cammino di lunghezza k, $k \geq 0$, dal vertice u al vertice v, allora u è detto essere un *ascendente* di v di *grado* k-esimo del vertice v, a sua volta, è detto essere un *discendente* di u di *grado* k-esimo. Così, il padre di un vertice v ne è un ascendente di primo grado di v, così come un figlio di v ne è un discendente di primo grado. Due vertici che non appartengono ad una stessa linea di discendenza sono detti *vertici collaterali*.

Un albero T è *ordinato* se, per ogni vertice v che non sia una foglia, esiste un ordinamento totale dei figli di v, talché è ben specificato, per ogni coppia di figli di v, chi dei due precede (o segue) l'altro. Un tale ordinamento induce il seguente ordinamento totale dell'intero insieme dei vertici di T. Siano u e v due qualsiasi vertici distinti di T. È sufficiente considerare il caso che u e v non siano figli di uno stesso vertice. I due vertici hanno sicuramente almeno un ascendente comune, che è la radice di T; per di più, è unico l'ascendente comune di massima profondità. Sia esso w. Se $w = u$ (o $w = v$), allora u precede (rispettivamente segue) v. Altrimenti, sia $(w, u', \ldots, u)$ il cammino che va da w a u e sia $(w, v', \ldots, v)$ il cammino che va da w a v; allora u precede (o segue) v se u' precede (rispettivamente, segue) v'. Consideriamo ad esempio l'albero ordinato T mostrato in figura



dove i vertici figli di uno stesso vertice sono così ordinati:

$$(b, c) \quad (d, e) \quad (f, g) \quad (h, i) \quad (m, n) .$$

L'ordinamento totale dei vertici di $\mathcal{T}$ è allora (a, b, d, f, h, l, m, n, i, g, e, c).

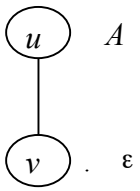
2.2 *Alberi di derivazione*

Sia $\boldsymbol{G} = (N, T, P, S)$ una grammatica acontestuale. Ricordiamo che una stringa x su T è una frase di $\boldsymbol{G}$ se $S \Rightarrow x$. Sia x una frase di $\boldsymbol{G}$. Una derivazione $(\sigma_0 = S, \sigma_1, \ldots, \sigma_k = x)$ di x da S può essere rappresentata graficamente con un albero ordinato. Per costruirlo, per ogni h, $0 \leq h \leq k$, associamo alla sottosequenza $(\sigma_0, \sigma_1, \ldots, \sigma_h)$ di $(\sigma_0, \sigma_1, \ldots, \sigma_k)$ un albero ordinato $\mathcal{T}_h$ così definito:

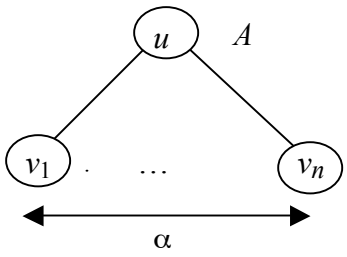
($h = 0$) $\mathcal{T}_0$ è un albero con un solo vertice etichettato con il simbolo iniziale di $\boldsymbol{G}$.

($0 < h \leq k$) Supponiamo che σ_h sia ottenuta da σ_{h-1} applicando la produzione $A \rightarrow \alpha$ per sostituire l'occorrenza di A che occupa la posizione i-esima nella stringa σ_{h-1}. Sia

u l'i-esima foglia di $\mathcal{T}_{h-1}$; così, il vertice u è etichettato con il simbolo nonterminale A. Distinguiamo due casi a seconda che la stringa α sia o meno vuota.
Caso 1: α è la stringa vuota ε. Allora, $\mathcal{T}_h$ è ottenuto da $\mathcal{T}_{h-1}$ aggiungendo un vertice v etichettato con ε.



Caso 2: α ha lunghezza $n > 0$. Allora, $\mathcal{T}_h$ è ottenuto da $\mathcal{T}_{h-1}$ aggiungendo n vertici v_1, …, v_n etichettati con i simboli della stringa α ed n archi (u, v_1), …, (u, v_n), e ordinando v_1, …, v_n nella stessa maniera in cui le loro etichette concorrono a formare la stringa α.



L'albero ordinato $\mathcal{T}_k$ che alla fine si ottiene è chiamato un *albero di derivazione* (parse tree) per la frase x di $\boldsymbol{G}$. Si osservi che x coincide con la stringa che si ottiene concatenando le etichette delle foglie di $\mathcal{T}_k$ prese nel giusto ordine. Ovviamente, gli alberi di derivazione associati a due distinte derivazioni di x non sono necessariamente distinti.

Esempio 2.1 Si consideri la grammatica acontestuale $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}, \mathtt{A}, \mathtt{B}\}$, $T = \{\mathtt{a}, \mathtt{b}\}$ e P contiene le seguenti produzioni:

$\mathtt{S} \rightarrow \mathtt{AB}$
$\mathtt{A} \rightarrow \mathtt{aA} \mid \mathtt{a}$
$\mathtt{B} \rightarrow \mathtt{bB} \mid \mathtt{b}$

La stringa `aabbb` è una frase di $\boldsymbol{G}$. Tre esempi di sue derivazioni dal simbolo iniziale di $\boldsymbol{G}$ sono

S, AB, AbB, AbbB, Abbb, aAbbb, aabbb
S, AB, aAB, aaB, aabB, aabbB, aabbb
S, AB, aAB, aAbB, aAbbB, aAbbb, aabbb

È facile vedere che i corrispondenti alberi di derivazione sono identici. ■

Esempio 2.2 Si consideri la grammatica acontestuale $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}\}$, $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ e P contiene le produzioni

$$\mathtt{S} \rightarrow \mathtt{SbS} \mid \mathtt{ScS} \mid \mathtt{a}$$

La stringa `abaca` è una frase di $\boldsymbol{G}$. Due esempi di sue derivazioni dal simbolo iniziale di $\boldsymbol{G}$ sono

$$\mathtt{S}, \mathtt{SbS}, \mathtt{abS}, \mathtt{abScS}, \mathtt{abSca}, \mathtt{abaca}$$

$$\mathtt{S}, \mathtt{ScS}, \mathtt{Sca}, \mathtt{SbSca}, \mathtt{abSca}, \mathtt{abaca}$$

È facile vedere che i corrispondenti alberi di derivazione sono distinti. ■

Viceversa, vedremo che, dato un albero di derivazione per una frase x di $\boldsymbol{G}$, in generale possiamo costruire più derivazioni due delle quali svolgono un ruolo importante nella compilazione.

Ricordiamo che il generico passo $\sigma_{h-1} \Rightarrow \sigma_h$ di una derivazione $(\sigma_0, \sigma_1, \ldots, \sigma_k)$ di x richiede l'esistenza di una produzione $A \rightarrow \alpha$ di $\boldsymbol{G}$ tale che esista almeno un'occorrenza del simbolo nonterminale A in σ_{h-1} per cui $\sigma_{h-1} = \sigma' A \sigma''$ e $\sigma_h = \sigma' \alpha \sigma''$. Ora, se una derivazione $(\sigma_0, \sigma_1, \ldots, \sigma_k)$ di x è tale che, per ogni $h \geq 0$, la scelta del simbolo nonterminale da sostituire e della sua occorrenza cade sempre sul simbolo nonterminale *più a sinistra* presente in σ_h, allora la derivazione è detta *sinistrorsa*. Analogamente, se la scelta è stata fatta prendendo sempre il simbolo nonterminale *più a destra* chiamiamo la derivazione *destrorsa*. Così, se $\sigma_{h-1} \Rightarrow \sigma_h$ è un passo di una derivazione sinistrorsa e $\sigma_{h-1} = \sigma' A \sigma''$, allora σ' è una stringa (eventualmente vuota) di simboli terminali; mentre, se $\sigma_{h-1} \Rightarrow \sigma_h$ è un passo di una derivazione destrorsa, allora è σ'' ad essere una stringa (eventualmente vuota) di simboli terminali. Ovviamente, se esiste una derivazione sinistrorsa (o destrorsa) di una stringa x, allora x è una frase di $\boldsymbol{G}$. Viceversa, se x è una frase di $\boldsymbol{G}$, allora esiste sempre una derivazione sinistrorsa ed una destrorsa di x. Infatti, se x è una frase di $\boldsymbol{G}$, allora esiste una derivazione di x. Sia $\mathcal{T}$ l'albero di derivazione associato. Mostriamo in maniera costruttiva che esiste una derivazione sinistrorsa $(\sigma_0, \sigma_1, \ldots, \sigma_k)$ di x. (In maniera analoga, si dimostra l'esistenza di una derivazione destrorsa di x.) La stringa σ_0 è (ovviamente) l'etichetta della radice (nodo 0) di $\mathcal{T}$ e la stringa σ_1 è la concatenazione delle etichette dei figli del nodo 0. Sia L la lista dei figli del nodo 0 che sono etichettati con simboli nonterminali. Se $L = \varnothing$, allora abbiamo finito perché $\sigma_1 = x$. Altrimenti, prendiamo il primo nodo in L. Sia esso il nodo i di $\mathcal{T}$ e sia A la sua etichetta. Allora, σ_2 è ottenuto da σ_1 sostituendo la prima occorrenza del simbolo A

con la concatenazione delle etichette dei figli del nodo i. Sia L' la lista dei figli del nodo i che sono etichettati con simboli nonterminali. Eliminiamo il nodo i da L ed aggiungiamo L' in testa alla lista L. Ripetiamo questa procedura fino a svuotare tutta la lista L e, a quel punto, avremo $\sigma_k = x$.
Va osservato infine che, dato un albero di derivazione $\mathcal{T}$ di x, le derivazioni sinistrorsa e destrorsa di x che si ottengono da $\mathcal{T}$ non sono necessariamente distinte.

Una frase x di $\boldsymbol{G}$ è *ambigua* se x ammette due distinti alberi di derivazione, ovvero se x ammette due distinte derivazioni sinistrorse (o destrorse). L'ambiguità di una frase può creare problemi interpretativi. L'esempio che segue dimostra che l'ambiguità può oscurare il significato di una frase (un'istruzione di assegnazione in un ipotetico linguaggio di programmazione).

Esempio 2.3 Si consideri la grammatica acontestuale $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove

$N = \{\mathtt{S}, \mathtt{A}, \mathtt{B}\}$ $\qquad\qquad$ $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{:=}, \mathtt{+}, \mathtt{*}, \mathtt{(}, \mathtt{)}\}$

e P contiene le produzioni

$\mathtt{S} \rightarrow \mathtt{A:=B}$
$\mathtt{A} \rightarrow \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$
$\mathtt{B} \rightarrow \mathtt{B+B} \mid \mathtt{B*B} \mid \mathtt{(B)} \mid \mathtt{A}$

Le frasi di $\boldsymbol{G}$ sono semplici istruzioni di assegnazione. La stringa `a:=b+c*a` è una frase di $\boldsymbol{G}$ ed ha due distinti alberi di derivazione. ■

Una grammatica acontestuale $\boldsymbol{G}$ è *ambigua* se esiste una frase di $\boldsymbol{G}$ che è ambigua. In alcuni casi, data una grammatica ambigua, se ne può trovare una equivalente che non è ambigua. Per esempio la grammatica ambigua

$\mathtt{S} \rightarrow \mathtt{A} \mid \mathtt{B}$
$\mathtt{A} \rightarrow \mathtt{a}$
$\mathtt{B} \rightarrow \mathtt{a}$

è equivalente alla grammatica non ambigua

$\mathtt{S} \rightarrow \mathtt{a}$

Un linguaggio acontestuale L è *intrinsecamente ambiguo* se ogni grammatica che genera L è ambigua. Ne è un esempio il linguaggio $L = \{\mathtt{a}^p\mathtt{b}^q\mathtt{c}^r : p = q \text{ o } q = r\}$ e la ragione intuitiva è che una qualsiasi grammatica generativa di L deve avere un tipo di albero di derivazione per le stringhe in cui $p = q$ ed un altro per le stringhe in cui $q = r$; sicché una stringa del tipo $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ avrà due distinti alberi di derivazione.

2.3 *Riconoscimento delle frasi*

In questo paragrafo daremo un algoritmo cha ha complessità polinomiale (per l'esattezza, cubica) per risolvere il problema del riconoscimento delle frasi di una grammatica acontestuale ***G***; cioè, per decidere se una data stringa di simboli terminali *x* è o meno una frase di ***G***. In caso affermativo, lo stesso algoritmo ci consentirà di costruire tutti gli alberi di derivazione per *x* e, quindi, di decidere se *x* è o meno una frase ambigua di ***G***.

A tale scopo, faremo uso della nozione di grammatica acontestuale in "forma normale di Chomsky" che andiamo a introdurre dopo aver dato alcune definizioni.

Sia ***G*** = (*N*, *T*, *P*, *S*) una grammatica acontestuale. Un simbolo nonterminale *A* di ***G*** diverso da *S* è

— una *variabile improduttiva di prima specie* se da *A* non è derivabile nessuna stringa di simboli terminali,

— una *variabile improduttiva di seconda specie* se dal simbolo iniziale *S* non è derivabile nessuna stringa che contenga *A*.

Esempio 2.4 Si consideri la grammatica ***G*** = (*N*, *T*, *P*, `S`) dove *N* = {`S`, `A`, `B`, `C`, `D`}, *T* = {`a`, `b`, `c`} e *P* contiene le seguenti produzioni:

`S` → `Aa` | `B` | `D`
`A` → `aA` | `bA` | `B`
`B` → `b`
`C` → `abc`

Qui `D` è una variabile improduttiva di prima specie e `C` è una variabile improduttiva di seconda specie. ■

D'ora in poi, data una produzionme $A \to \alpha$, chiamiamo il simbolo nonterminale *A* la *testa* della produzione e la stringa α il *corpo* della produzione.

Una produzione di ***G*** è

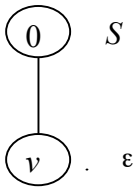
— una *produzione inutile* se contiene una variabile improduttiva,
— una *produzione nulla* se è della forma $A \to \varepsilon$,

La grammatica ***G*** è in *forma normale di Chomsky* se è priva di produzioni inutili e, inoltre, se

(*i*) l'unica produzione nulla di ***G*** eventualmente presente è $S \to \varepsilon$ e, se lo è, allora *S* non compare nel corpo di nessuna produzione di ***G***,

(*ii*) ogni produzione nonnulla è in una delle due forme: $A \to a$ o $A \to BC$.

Vale la pena osservare che, data una grammatica $\boldsymbol{G}$ in forma normale di Chomsky, se la stringa vuota ε è una frase di $\boldsymbol{G}$, allora (S, ε) è l'unica derivazione di ε, e se una stringa nonvuota x è una frase di $\boldsymbol{G}$, allora in ogni derivazione di x ad ogni passo andremo ad applicare una produzione nonnulla della forma $A \rightarrow a$ oppure $A \rightarrow BC$. in altri termini, se ε è una frase di $\boldsymbol{G}$, allora abbiamo un solo albero di derivazione per ε



mentre, se una stringa nonvuota x è una frase di $\boldsymbol{G}$, allora ogni albero di derivazione per x è un albero binario (vale a dire, ogni vertice ha al più due archi uscenti) e nessuno dei suoi vertici è etichettato con la stringa vuota.

Daremo ora una procedura di normalizzazione che trasforma in forma normale di Chomsky una qualsiasi grammatica acontestuale data.

Cominciamo con due algoritmi che eliminano rispettivamente variabili improduttive di prima specie e variabili improduttive di seconda specie con relative produzioni.

Algoritmo 2.1

INPUT: una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$.
OUTPUT: una grammatica $\boldsymbol{G}' = (N', T, P', S)$ equivalente a $\boldsymbol{G}$ e priva di variabili improduttive di prima specie.

PROCEDURA

0. $N' := \{S\}$

1. $N' := N' \cup \{A \in N : \exists\, A \rightarrow \alpha \text{ in } P \text{ tale che } \alpha \in T^*\}$.
2. $P' := \{A \rightarrow \alpha \in P : A \in N' \text{ e } \alpha \in T^*\}$.
3. Finché l'insieme N' non possa essere ulteriormente ampliato, ripetere

$N' := N' \cup \{A \in N \backslash N' : \exists\, A \rightarrow \alpha \text{ in } P \text{ tale che } \alpha \in (N' \cup T)^+\}$

$P' := P' \cup \{A \rightarrow \alpha \in P \backslash P' : A \in N' \text{ ed } \alpha \in (N' \cup T)^+\}$.

Algoritmo 2.2

INPUT: una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$.
OUTPUT: una grammatica $\boldsymbol{G}' = (N', T', P', S)$ equivalente a $\boldsymbol{G}$ e priva di variabili improduttive di seconda specie.

PROCEDURA
1. $N' := \{S\}$, $T' := \varnothing$, $P' := \varnothing$.

2. Finché l'insieme P' non possa essere ulteriormente ampliato, ripetere

per ogni $A \in N'$
per ogni $A \to \alpha$ in P
$T' := T' \cup \{t \in T \backslash T' : t \text{ compare in } \alpha\}$;
$N' := N' \cup \{B \in N \backslash N' : B \text{ compare in } \alpha\}$;
aggiungere a P' la produzione $A \to \alpha$.

Esempio 2.4 (seguito). L'applicazione dell'Algoritmo 2.1 alla grammatica $\boldsymbol{G}$ produce la grammatica $\boldsymbol{G}' = (N', T, P', \mathtt{S})$, dove $N' = \{\mathtt{S}, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$ e P' contiene le seguenti produzioni:

$\mathtt{S} \to \mathtt{Aa} \mid \mathtt{B}$
$\mathtt{A} \to \mathtt{aA} \mid \mathtt{bA} \mid \mathtt{B}$
$\mathtt{B} \to \mathtt{b}$
$\mathtt{C} \to \mathtt{abc}$

L'applicazione dell'Algoritmo 2.2 alla grammatica $\boldsymbol{G}'$ produce la grammatica $\boldsymbol{G}'' = (N'', T'', P'', \mathtt{S})$, dove $N'' = \{\mathtt{S}, \mathtt{A}, \mathtt{B}\}$, $T'' = \{\mathtt{a}, \mathtt{b}\}$ e P'' contiene le seguenti produzioni:

$\mathtt{S} \to \mathtt{Aa} \mid \mathtt{B}$
$\mathtt{A} \to \mathtt{aA} \mid \mathtt{bA} \mid \mathtt{B}$
$\mathtt{B} \to \mathtt{b}$

Esempio 2.5 Si consideri la grammatica $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}, \mathtt{A}, \mathtt{B}\}$, $T = \{\mathtt{a}\}$ e P contiene le seguenti produzioni:

$\mathtt{S} \to \mathtt{AB} \mid \mathtt{a}$
$\mathtt{A} \to \mathtt{a}$

L'applicazione dell'Algoritmo 2.1 e quindi dell'Algoritmo 2.2 produce la grammatica $\boldsymbol{G}' = (N', T, P', \mathtt{S})$ dove $N' = \{\mathtt{S}\}$ e P' contiene l'unica produzione $\mathtt{S} \to \mathtt{a}$. Invece, invertendo l'ordine di applicazione dei due algoritmi, avremo la grammatica $\boldsymbol{G}'' = (N'', T'', P'', \mathtt{S})$ dove $N'' = \{\mathtt{S}, \mathtt{A}\}$ e P'' contiene le due produzioni: $\mathtt{S} \to \mathtt{a}$ ed $\mathtt{A} \to \mathtt{a}$. ■

Una volta eliminate le produzioni inutili, otterremo una grammatica in forma norma di Chomsky dopo aver eliminato le produzioni nulle, quindi le *produzioni unitarie* (cioè quelle della forma $A \to B$) e, infine, le *produzioni multiple* (cioè quelle della forma A

$\rightarrow \alpha$, dove α è una stringa di lunghezza maggiore di 2 oppure è della forma aB o Ba).

Comiciamo dall'eliminazione delle produzioni nulle. A tale scopo, diciamo che un simbolo nonterminale è una *variabile annullabile* se da esso è derivabile la stringa vuota. L'algoritmo che segue costruisce una grammatica $\boldsymbol{G'}$ equivalente a $\boldsymbol{G}$ con al più una produzione nulla. In esso, l'insieme delle variabili annullabili di $\boldsymbol{G}$ è indicato con E.

Algoritmo 2.3

INPUT: una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$.
OUTPUT: una grammatica $\boldsymbol{G'}$ equivalente a $\boldsymbol{G}$.

PROCEDURA

1. $E := \{A \in N: A \rightarrow \varepsilon \in P\}$.

2. Finché l'insieme E non possa essere ulteriormente ampliato, ripetere:

   per ogni $B \in N \backslash E$, se esiste una produzione $B \rightarrow X$ in P dove X è una stringa nonvuota di sole variabili annullabili, allora aggiungere B ad E.

3. $P' := \varnothing$.
   Per ogni produzione nonnulla $A \rightarrow \alpha$ in P,

   se α non contiene nessuna variabile annullabile, allora aggiungere a P' la produzione $A \rightarrow \alpha$,

   altrimenti, indicate con $i_1, \ldots, i_k$ le posizioni occupate dalle variabili annullabili che sono presenti in α,

   per ogni sottoinsieme (proprio e improprio) I di $\{i_1, \ldots, i_k\}$, ripetere:

   costruire la stringa β che si ottiene da α eliminando le variabili annullabili che occupano le posizioni contenute in I;

   se $\beta \neq \varepsilon$, allora aggiungere $A \rightarrow \beta$ a P'.

4. Applicare l'Algoritmo 2.1 per eliminare le eventuali variabili improduttive di prima specie e le relative produzioni inutili.

5. Se S è una variabile annullabile (cioè se $S \in E$), allora

se S non compare nel corpo di nessuna produzione in P', allora
aggiungere a P' la produzione nulla $S \to \varepsilon$;
$\boldsymbol{G}' := (N, T, P', S)$;

altrimenti
$N' := N \cup \{S'\}$;
aggiungere a P' le due produzioni $S' \to S \mid \varepsilon$;
restituire $\boldsymbol{G}' = (N', T, P', S')$.

Esempio 2.6 Si consideri la grammatica acontestuale $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}, \mathtt{A}, \mathtt{B}\}$, $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ e P contiene le produzioni

$\mathtt{S} \to \mathtt{aS} \mid \mathtt{AB}$
$\mathtt{A} \to \mathtt{bAc} \mid \varepsilon$
$\mathtt{B} \to \varepsilon$

Se applichiamo l'Algoritmo 2.3 alla grammatica $\boldsymbol{G}$, dopo aver eseguito l'Istruzione 2 abbiamo $E = \{\mathtt{S}, \mathtt{A}, \mathtt{B}\}$, e dopo aver eseguito l'Istruzione 3, abbiamo

$P' = \{\mathtt{S} \to \mathtt{a} \mid \mathtt{aS} \mid \mathtt{A} \mid \mathtt{B} \mid \mathtt{AB},\ \mathtt{A} \to \mathtt{bc} \mid \mathtt{bAc}\}$.

A questo punto, eliminiamo B perché è una variabile improduttiva di prima specie.

$P' = \{\mathtt{S} \to \mathtt{a} \mid \mathtt{aS} \mid \mathtt{A},\ \mathtt{A} \to \mathtt{bc} \mid \mathtt{bAc}\}$.

Aggiungiamo infine le produzioni $\mathtt{S}' \to \mathtt{S} \mid \varepsilon$, sicché l'Algoritmo 2.3 termina restituendo $\boldsymbol{G}' = (N', T, P', \mathtt{S}')$ dove

$N' = \{\mathtt{S}', \mathtt{S}, \mathtt{A}\}$
$P' = \{\mathtt{S}' \to \mathtt{S} \mid \varepsilon, \mathtt{S} \to \mathtt{a} \mid \mathtt{aS} \mid \mathtt{A},\ \mathtt{A} \to \mathtt{bc} \mid \mathtt{bAc}\}$. ■

L'algoritmo che segue costruisce una grammatica $\boldsymbol{G}'$ equivalente a $\boldsymbol{G}$ priva di produzioni unitarie.

Algoritmo 2.4

INPUT: una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$.
OUTPUT: una grammatica $\boldsymbol{G}'$ equivalente a $\boldsymbol{G}$.

PROCEDURA
1. Per ogni $A \in N$,

$\overline{A} := \{A\}$;
finché l'insieme $\overline{A}$ non possa essere ulteriormente ampliato,
ripetere

per ogni $B \in N \setminus \overline{A}$, se esiste $C \to B$ in P con $C \in \overline{A}$
allora aggiungere B ad $\overline{A}$.

2. $N' := N$ e $P' := P$.
Per ogni $A \in N$ tale che $\overline{A} \neq \{A\}$,
per ogni $B \in \overline{A} \setminus \{A\}$,
per ogni produzione nonunitaria $B \to \beta$ in P' aggiungere
$A \to \beta$ a P'.

3. Eliminare da P' tutte le produzioni unitarie.

4. Applicare l'Algoritmo 2.2 per eliminare le eventuali variabili improduttive di seconda specie e le relative produzioni inutili.

Esempio 2.6 (seguito) Quando applichiamo l'Algoritmo 2.4, dopo aver eseguito l'Istruzione 1 abbiamo

$\overline{\mathtt{S}'} = \{\mathtt{S}', \mathtt{S}, \mathtt{A}\}$ $\overline{\mathtt{S}} = \{\mathtt{S}, \mathtt{A}\}$ $\overline{\mathtt{A}} = \{\mathtt{A}\}$

Dopo aver eseguito l'Istruzione 2, abbiamo le produzioni

$\mathtt{S}' \to \mathtt{S} \mid \varepsilon$ $\mathtt{S}' \to \mathtt{a} \mid \mathtt{aS}$ $\mathtt{S}' \to \mathtt{bc} \mid \mathtt{bAc}$
$\mathtt{S} \to \mathtt{a} \mid \mathtt{aS} \mid \mathtt{A}$ $\mathtt{S} \to \mathtt{bc} \mid \mathtt{bAc}$
$\mathtt{A} \to \mathtt{bc} \mid \mathtt{bAc}$

Dopo aver eseguito l'Istruzione 3, abbiamo le sole produzioni

$\mathtt{S}' \to \mathtt{a} \mid \mathtt{aS} \mid \mathtt{bc} \mid \mathtt{bAc} \mid \varepsilon$
$\mathtt{S} \to \mathtt{a} \mid \mathtt{aS} \mid \mathtt{bc} \mid \mathtt{bAc}$
$\mathtt{A} \to \mathtt{bc} \mid \mathtt{bAc}$

■

L'algoritmo che segue costruisce una grammatica $\boldsymbol{G}'$ equivalente a $\boldsymbol{G}$ priva di produzioni multiple.

Algoritmo 2.5

INPUT: una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$.
OUTPUT: una grammatica $\boldsymbol{G}'$ equivalente a $\boldsymbol{G}$.

PROCEDURA
1. $N' := N$ e $P' := P$.

2. Se a è un simbolo terminale tale che esista in P' una produzione $A \to \alpha$ in cui compare a e α è una stringa di lunghezza maggiore di uno, allora introdurre un nuovo simbolo nonterminale C_a, aggiungere C_a ad N' ed aggiungere $C_a \to a$ a P'. Quindi, per ogni produzione $A \to \alpha$ in P' in cui α è una stringa di lunghezza maggiore di 1 che contenga almeno un simbolo terminale,

   costruire la stringa X che si ottiene da α sostituendo ad ogni simbolo terminale a il simbolo nonterminale C_a,

   eliminare $A \to \alpha$ da P' e aggiungere $A \to X$ a P'.

3. Per ogni produzione $A \to B_1 \ldots B_k$ in P' con $k > 2$ e $B_i \in N'$ $(1 \le i \le k)$,

   introdurre k–2 nuovi simboli nonterminali $C_1, \ldots, C_{k-2}$:
   $N' := N' \cup \{C_1, \ldots, C_{k-2}\}$;
   eliminare $A \to B_1 \ldots B_k$ da P';
   aggiungere a P' le k–1 produzioni:

   $$A \to B_1 C_1 \qquad C_1 \to B_2 C_2 \qquad \ldots \qquad C_{k-2} \to B_{k-1} B_k.$$

4. Restituire $\boldsymbol{G}' = (N', T, P', S)$.

Data una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$ in forma normale di Chomsky, per decidere se una stringa nonvuota x di simboli terminali è o meno una frase di $\boldsymbol{G}$, possiamo procedere alla maniera seguente. Innanzitutto, se x è la stringa vuota, allora x è una frase di $\boldsymbol{G}$ se e solo se $S \to \varepsilon$ è una produzione di $\boldsymbol{G}$. Consideriamo ora il caso che x abbia lunghezza $n > 0$, e indichiamo con $x_{i,l}$ la sottostringa x che inizia alla posizione i-esima ed ha lunghezza l, $i = 1, \ldots, n$ e $l = 1, \ldots, n$–i+1. Sia $N_{i,l}$ l'insieme dei simboli nonterminali di $\boldsymbol{G}$ da cui $x_{i,l}$ è derivabile; cioè,

$$N_{i,l} = \{A \in N : A \stackrel{*}{\Rightarrow} x_{i,l}\}.$$

Si noti che $x_{1,n} = x$ cosicché x è una frase di $\boldsymbol{G}$ se e solo se S appartiene ad $N_{1,n}$. Gli insiemi $N_{i,l}$ possono essere riportati in una tabella indicizzata dalla coppia (i, l) la quale non è definita al di sotto della diagonale che va dalla cella $(n, 1)$ alla cella $(1, n)$, vale a dire che la riga i-esima $(1 \le i \le n)$ è definita fino alla colonna $(n$–i+1)-esima. Si osservi che il numero di tali celle è pari a $\dfrac{n(n+1)}{2}$. Vediamo come possono essere calcolati gli elementi di questa tabella. Per le celle della prima colonna ($1 \le i \le n$ e $l = 1$), la cosa è semplice visto che $A \stackrel{*}{\Rightarrow} x_{i,1}$ se e solo se $\boldsymbol{G}$ contiene la produzione $A \to x_{i,1}$; pertanto, possiamo porre

$$N_{i,1} := \{A \in N : A \to x_{i,1} \text{ è in } P\}.$$

Consideriamo ora una cella (i, l) con $l > 1$ $(1 \leq i \leq n–l+1)$. Siccome $\boldsymbol{G}$ è in forma normale di Chomsky e la lunghezza di $x_{i,l}$ è $l > 1$, il simbolo nonterminale A appartiene ad $N_{i,l}$ se e solo se $\boldsymbol{G}$ contiene una produzione $A \rightarrow BC$ tale che $x_{i,l}$ risulti dalla concatenazione di due stringhe y e z che siano derivabili rispettivamente da B e da C; cioè, $A \stackrel{*}{\Rightarrow} x_{i,l}$ se e solo se esiste una produzione $A \rightarrow BC$ tale che

$$B \stackrel{*}{\Rightarrow} y \qquad C \stackrel{*}{\Rightarrow} z \qquad x_{i,l} = yz.$$

Una tale coppia di sottostringhe di $x_{i,l}$ esiste se e solo se esiste j, $1 \leq j \leq l–1$, per cui $y = x_{i,j}$ e $z = x_{i+j,l–j}$. Dunque, A appartiene a $N_{i,l}$ se e solo se esiste una produzione $A \rightarrow BC$ ed esiste j, $1 \leq j \leq l–1$, per cui $B \in N_{i,j}$ e $C \in N_{i+j,l–j}$.

Riportiamo per comodità i due casi $j < l/2$ e $j > l/2$.

…	…	…	…	…
i,j $x_{i,j}$	…	…	…	i,l $x_{i,l}$
…	…	…	…	…
…	…	$i+j,l–j$ $x_{i+j,l–j}$	…	…
…	…	…	…	…

…	…	…	…	…
i,j B	…	…	…	i,l A
…	…	…	…	…
…	…	$i+j,l–j$ C	…	…
…	…	…	…	…

Caso $j < l/2$

…	…	…	…	…
…	…	i,j $x_{i,j}$	…	i,l $x_{i,l}$
…	…	…	…	…
$i+j,l–j$ $x_{i+j,l–j}$	…	…	…	…
…	…	…	…	…

…	…	…	…	…
…	…	i,j B	…	i,l A
…	…	…	…	…
$i+j,l–j$ C	…	…	…	…
…	…	…	…	…

Caso $j > l/2$

Si osservi che le celle (i, j) e $(i+j, l–j)$ si trovano sempre in due colonne che sono a sinistra della colonna l-esima. Pertanto, $N_{i,l}$ può essere calcolato se le colonne 1, 2, …, l–1 sono state già riempite. Questa osservazione è alla base del seguente algoritmo, che è dovuto a Cocke, Younger e Kasami ed è comunemente citato come *algoritmo CYK*.

Algoritmo CYK

INPUT: una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$ in forma normale di Chomsky, una stringa x su T di lunghezza n e le sottostringhe $x_{1,1}, \ldots, x_{n,1}$ di x.

OUTPUT: il valore della variabile booleana *test* uguale a VERO se e solo se x appartiene al linguaggio generato da $\boldsymbol{G}$.

PROCEDURA

1. *test* := FALSO.

2. Se $n = 0$, allora
   se P contiene la produzione $S \rightarrow \varepsilon$, allora *test* := VERO e Uscire.

3. Per $i = 1, \ldots, n$
   $N_{i,1} := \varnothing$;
   per ogni $A \in N$, se P contiene la produzione $A \rightarrow x_{i,1}$, allora aggiungere A a $N_{i,1}$.

4. Per $l = 2, \ldots, n$
   per $i = 1, \ldots, n–l+1$
   $N_{i,l} := \varnothing$;
   per $j = 1, \ldots, l–1$
   per ogni produzione in P della forma $A \rightarrow BC$
   se $B \in N_{i,j}$ e $C \in N_{i+j,l–j}$, allora aggiungere A a $N_{i,l}$.

5. Se $S \in N_{1,n}$, allora *test* := VERO.

Esempio 2.7 Si consideri la grammatica acontestuale $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$, $T = \{\mathtt{a}, \mathtt{b}\}$ e P contiene le seguenti produzioni:

$\mathtt{S} \rightarrow \mathtt{AB} \mid \mathtt{BC}$
$\mathtt{A} \rightarrow \mathtt{BA} \mid \mathtt{a}$
$\mathtt{B} \rightarrow \mathtt{CC} \mid \mathtt{b}$
$\mathtt{C} \rightarrow \mathtt{AB} \mid \mathtt{a}$

Vogliamo decidere se la stringa x = `bbab` è o meno una frase di $\boldsymbol{G}$. Le sottostringhe $x_{i,l}$ di x sono riportate qui di seguito.

^{11} `b`	^{12} `bb`	^{13} `bba`	^{14} `bbab`
^{21} `b`	^{22} `ba`	^{23} `bab`	-
^{31} `a`	^{32} `ab`	-	-
^{41} `b`	-	-	-

Tabella delle sottostringhe $x_{i,l}$

Con l'algorimo CYK calcoliamo gli insiemi $N_{i,l}$.

^{11} `B`	^{12} $\varnothing$	^{13} `A`	^{14} `S, C`
^{21} `B`	^{22} `S, A`	^{23} `S, C`	-
^{31} `A, C`	^{32} `S, C`	-	-
^{41} `B`	-	-	-

Tabella degli insiemi $N_{i,l}$

Visto che `S` appartiene ad $N_{1,4}$, possiamo concludere che la stringa `bbab` è una frase di $\boldsymbol{G}$. ■

Nel caso che la stringa x sia una frase della grammatica, possiamo costruire tutti i possibili alberi di derivazione per x ancora con l'Algoritmo CYK se ad ogni cella (i, l) associamo una foresta $\mathcal{F}(i, l)$ di alberi ordinati, inizialmente vuota. Ogni albero di queste foreste ha due o tre vertici a secondo del suo tipo: un albero è del *primo tipo* se è della forma

padre: figlio

ed è del *secondo tipo* se è della forma

padre: primo figlio, secondo figlio.

In ogni albero del primo tipo, il vertice-padre è etichettato con un simbolo nonterminale mentre il vertice-figlio è etichettato con un simbolo terminale. In ogni albero del secondo tipo, i vertici sono tutti etichettati con simboli nonterminali; inoltre, ognuno dei due figli ha associata una cella (i, l) che chiamiamo *indice* di quel vertice. Infine, diciamo che un albero è un *A-albero* se A è l'etichetta del suo vertice-padre.

Qui di seguito sono riportate le modifiche delle istruzioni 3 e 4 dall'algoritmo CYK.

3. Per $i = 1, \ldots, n$

…

se P contiene la produzione $A \rightarrow x_{i,1}$, allora

aggiungere A a $N_{i,1}$;

aggiungere ad $\mathcal{F}(i, 1)$ un A-albero del primo tipo con il vertice-figlio etichettato con il simbolo terminale $x_{i,1}$.

4. Per $l = 2, \ldots, n$

…

per ogni produzione in P della forma $A \rightarrow BC$

se $B \in N_{i,j}$ e $C \in N_{i+j,l-j}$, allora

aggiungere A a $N_{i,l}$;

aggiungere ad $\mathcal{F}(i, l)$ un A-albero del secondo tipo in cui il primo figlio ha etichetta B e indice (i, j) ed il secondo figlio ha etichetta C e indice $(i+j, l-j)$.

Se x è una frase di $\boldsymbol{G}$, gli alberi di derivazione per x si ottengono sviluppando ciascuno degli S-alberi della foresta $\mathcal{F}(1, n)$. Vediamo come. Sia $\mathcal{T}$ uno di questi. L'albero $\mathcal{T}$ è un albero di secondo tipo. Siano v_1 e v_2 le foglie di $\mathcal{T}$, A_1 e A_2 le loro etichette e (i_1, l_1) e (i_2, l_2) i loro indici. Si osservi che la foresta $\mathcal{F}(i_1, l_1)$ certamente conterrà uno o più A_1-alberi. Sia $\mathcal{T}_1$ uno di questi. Andremo allora a sostituire alla foglia v_1 di $\mathcal{T}$ l'albero $\mathcal{T}_1$ e ripeteremo questa operazione per ognuno degli A_1-alberi della foresta $\mathcal{F}(i_1, l_1)$. Si otterrà così una foresta di S-alberi, in ognuno dei quali il vertice v_2 è ancora una foglia. A questo punto, espandiamo v_2 così come abbiamo fatto per v_1. E così via finché tutte le foglie non saranno etichettate con simboli terminali. Ripetendo questa operazione per ogni S-albero di $\mathcal{F}(1, n)$, otterremo alla fine tutti i possibili alberi di derivazione per x.

Esempio 2.7 (seguito) L'applicazione della nostra variante dell'algoritmo CYK associa alle dieci celle le seguenti foreste:

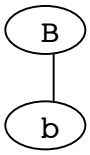
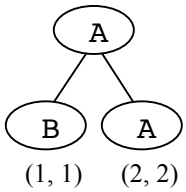
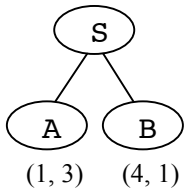
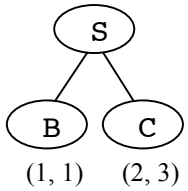
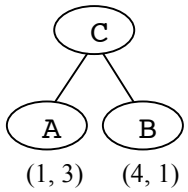
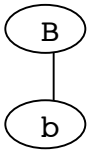
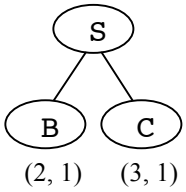
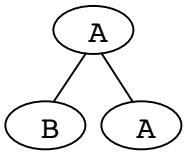
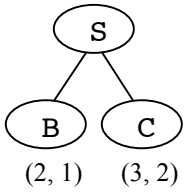
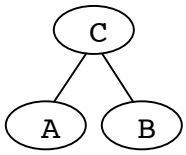
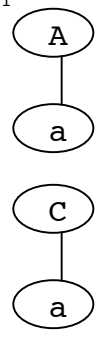
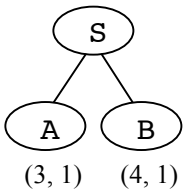
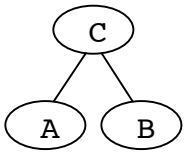
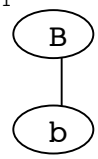
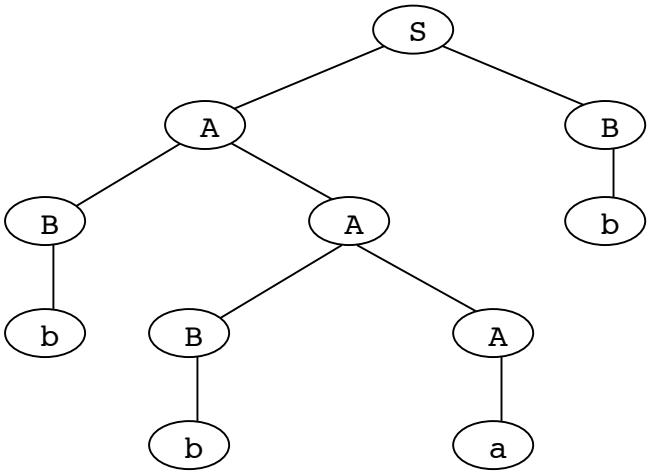
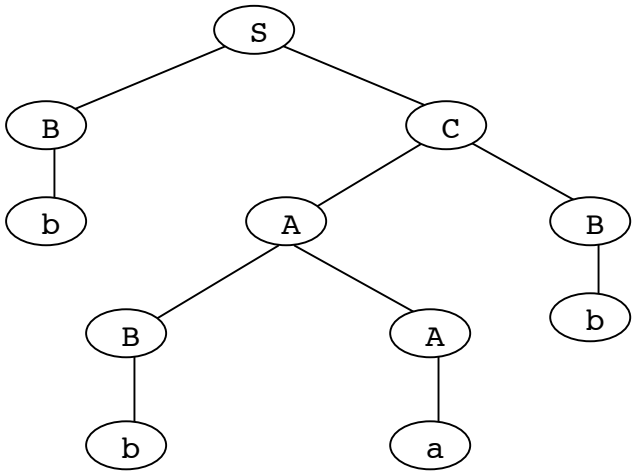
11  	12	13  (1, 1) (2, 2)	14  (1, 3) (4, 1)  (1, 1) (2, 3)  (1, 3) (4, 1)
21  	22  (2, 1) (3, 1)  (2, 1) (3, 1)	23  (2, 1) (3, 2)  (2, 2) (4, 1)	-
31  	32  (3, 1) (4, 1)  (3, 1) (4, 1)	-	-
41  	-	-	-

Tabella delle foreste $F(i, l)$

Ora, sviluppando il primo S-albero di $\mathcal{F}(1, 4)$ otteniamo l'albero di derivazione:



e sviluppando il secondo S-albero di $\mathcal{F}(1, 4)$ otteniamo l'albero di derivazione:



Questi sono gli unici alberi di derivazione per la stringa bbab dal simbolo iniziale della grammatica ***G***, che pertanto risulta essere ambigua. ■

Cap. 3

LINGUAGGI DI PROGRAMMAZIONE

Un linguaggio di programmazione è definito dalle sue componenti grammaticale e semantica. Come in ogni lingua, la componente grammaticale di un linguaggio di programmazione consiste di una parte lessicale ed una sintattica: la prima specifica le parole o *lessemi* (come il sostantivo, l'articolo, l'aggettivo, etc. nella lingua naturale); la seconda le parti del discorso o *sintagmi* (come la proposizione, il soggetto, il predicato etc. nella lingua naturale). I lessemi sono raggruppati in *classi lessicali* in base alla loro struttura morfologica e i sintagmi in costrutti sintattici o *categorie sintattiche*. Sia le classi lessicali che le categorie sintattiche sono identificate da nomi propri. Chiamiamo *figure lessicali* (tokens) i nomi delle classi lessicali e le indicheremo in grassetto. I *valori* di una figura lessicale **t** sono gli elementi (i lessemi) della classe lessicale di nome **t**. Una figura lessicale è una *variabile lessicale* se ha due o più valori; altrimenti, è una *costante lessicale*.
Specificata la componente grammaticale di un linguaggio di programmazione, è ben definito cos'è un *programma*. Qual è poi il risultato dell'esecuzione di un programma è specificato dalla componente semantica del linguaggio di programmazione, di cui parleremo nel capitolo 9.

3.1 Grammatiche lessicali

Le classi lessicali di un linguaggio di programmazione sono tutte generabili da grammatiche regolari di cui le figure lessicali sono i simboli iniziali. Diamo alcuni esempi per un ipotetico linguaggio di programmazione. Siano

> **id** la figura lessicale che denomina la classe lessicale degli identificatori,
> **num** la figura lessicale denomina la classe lessicale dei numeri (reali) assoluti (cioè senza segno),
> **comp** la figura lessicale che denomina la classe lessicale degli operatori comparativi,
> **add** e **multop** le figure lessicali che denominano le classi lessicali delle operazioni (additive e moltiplicative).

— Produzioni della grammatica lessicale generativa degli identificatori:

id	→	**lettera coda**
coda	→	**lettera coda** | **cifra coda** | ε
lettera	→	`A` | `B` | … | `Z` | `a` | `b` | … | `z`
cifra	→	`0` | `1` | … | `9`

— Produzioni della grammatica lessicale generativa dei numeri assoluti:

num	→	**intero decimale esponente**
sequenza	→	**cifra sequenza** | ε
intero	→	**cifra sequenza**
decimale	→	**.** **cifra sequenza** | ε
esponente	→	`E` **segno cifra sequenza** | ε
segno	→	`+` | `–` | ε
cifra	→	`0` | `1` | **...** | `9`

— Produzioni della grammatica lessicale generativa degli operatori comparativi:

comp	→	`<` | `<=` | `=` | `<>` | `>` | `>=`

— Produzioni della grammatica lessicale generativa delle operazioni di addizione e moltiplicazione:

add	→	`+` | `–` | `or`
mol	→	`*` | `/` | `div` | `mod` | `and`

Quelle che seguono sono alcune tipiche costanti lessicali di un linguaggio di programmazione:

— parole chiave

pgm	→	`program`
var	→	`var`
array	→	`array`
of	→	`of`
integer	→	`integer`
real	→	`real`
function	→	`function`
procedure	→	`procedure`
begin	→	`begin`
end	→	`end`
if	→	`if`
then	→	`then`
else	→	`else`
while	→	`while`
do	→	`do`
const	→	`const`

— l'operazione di assegnazione

ass → **:=**

— l'operazione di esponenziazione

exp → ******

— le parentesi

pts → **(**
ptd → **)**
pqs → **[**
pqd → **]**
…

— i segni di interpunzione

pv → **;**
pp → **..**
…

Vedremo più in là una notazione compatta per esprimere le produzioni di una grammatica lessicale che fa uso delle cosiddette *espressioni regolari* (v. capitolo 5). Ad esempio, la grammatica generativa dei numeri assoluti viene definita alla maniera seguente:

num → **intero decimale esponente**
intero → **cifra cifra***
decimale → **.** **cifra cifra*** | ε
esponente → E (+ | – | ε) **cifra cifra*** | ε
cifra → [0 1…9]

o, ancora, più semplicemente

num → **intero decimale esponente**
intero → **cifra**$^+$
decimale → **.** **cifra**$^+$ | ε
esponente → E (+ | – | ε) **cifra**$^+$ | ε
cifra → [0-9]

fino alla sua forma definitiva

num → **sequenza** (**.** **sequenza**)? (E [+–]? **sequenza**)?

sequenza → **cifra**$^{+}$
cifra → [0-9]

Consideriamo ad esempio un testo che contenga la seguente istruzione nel linguaggio di programmazione Pascal:

```
costo := canone + consumo * 10        (1)
```

In questo testo compaiono nell'ordine i seguenti lessemi:

`costo`	è un valore della figura lessicale **id**
`:=`	è il valore della figura lessicale **ass**
`canone`	è un valore della figura lessicale **id**
`+`	è un valore della figura lessicale **add**
`consumo`	è un valore della figura lessicale **id**
`*`	è un valore della figura lessicale **mol**
`10`	è un valore della figura lessicale **num**

La concatenazione delle corrsispondenti figure lessicali

id ass id add id mol num (2)

è quello che chiamiamo lo *schema lessicale* del testo (1)

3.2 Grammatiche sintattiche

Le categorie sintattiche di un linguaggio di programmazione sono specificate come simboli nonterminali di un'unica grammatica ***G*** che nella maggior parte dei casi è una grammatica acontestuale (anche se in taluni casi, come nei linguaggi C e Java, essa è una grammatica contestuale). I simboli terminali di ***G*** sono le figure lessicali del linguaggio di programmazione ed il simbolo iniziale di ***G*** è la variabile che ne rappresenta i programmi. Ecco un frammento di ***G*** per un ipotetico linguaggio di programmazione:

program → **pgm id** (*L*) **;** *D R I* **.**

…

B → *I* | *B* **;** *I*

I → **id ass** *E*
| **if** *E* **then** *I*
| **if** *E* **then** *I* **else** *I*
| **while** *E* **do** *I*
| **begin** *I* **end**

$E \rightarrow S \mid S$ **comp** S

$S \rightarrow T \mid S$ **add** T

$T \rightarrow$ **id** | **num** | T **mol** F

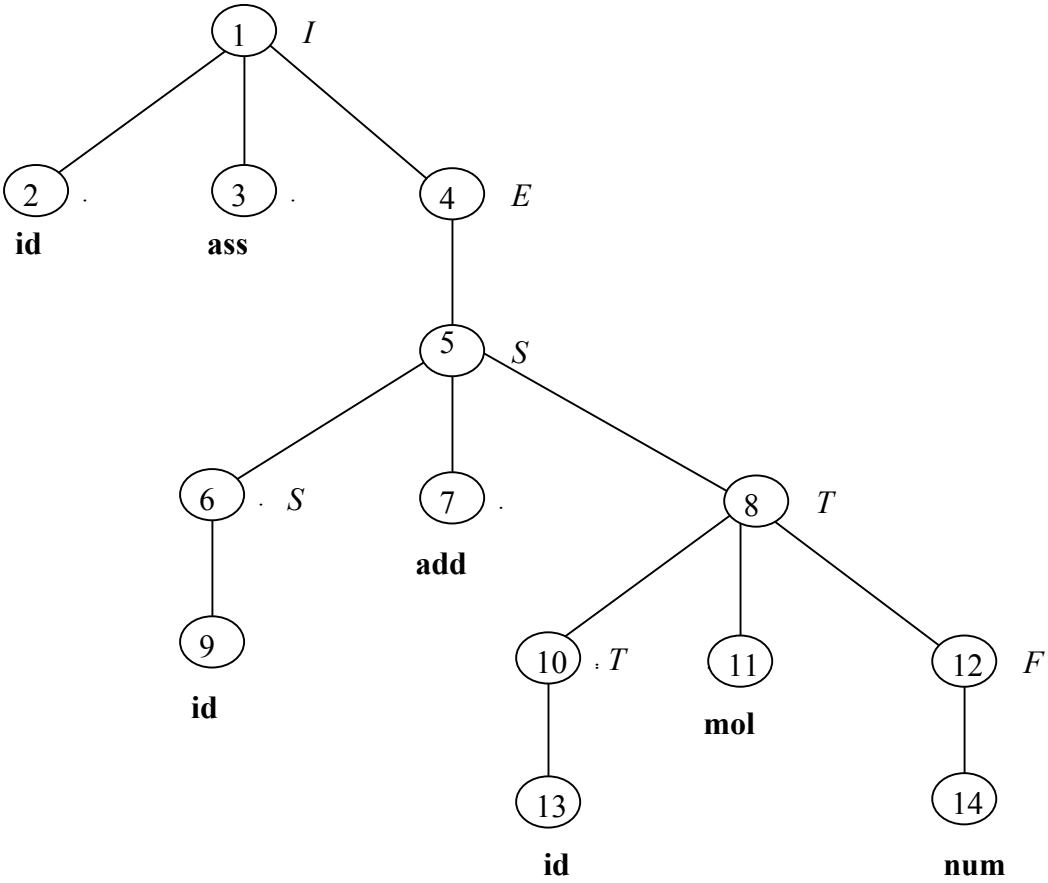
$F \rightarrow$ **id** | **num** | F **exp** F

…

Qui

la variabile *program* rappresenta la categoria sintattica dei programmi,
la variabile *L* rappresenta la categoria sintattica delle liste di identificatori,
la variabile *D* rappresenta la categoria sintattica delle dichiarazioni di sottoprogrammi,
la variabile *B* rappresenta la categoria sintattica dei blocchi di istruzioni,
la variabile *I* rappresenta la categoria sintattica delle istruzioni,
la variabile *E* rappresenta la categoria sintattica delle espressioni,
la variabile *S* rappresenta la categoria sintattica delle espressioni semplici,
la variabile *T* rappresenta la categoria sintattica dei termini,
la variabile *F* rappresenta la categoria sintattica dei fattori.

Consideriamo di nuovo il testo (1). Il suo schema lessicale (3) è una stringa derivabile dal simbolo nonterminale *I* come dimostra l'albero di derivazione mostrato in figura


1
I
2
id
3
ass
4
E
5
S
6
S
7
add
8
T
9
id
10
T
11
mol
12
F
13
id
14
num

Cap. 4

COMPILAZIONE

Un *compilatore* è un programma che legge una stringa di caratteri (un *testo*) su un dato alfabeto e verifica che essa sia un programma in un assegnato linguaggio di programmazione (il *linguaggio sorgente*) come il Fortran, il Pascal etc.; in caso affermativo, il testo è chiamato il *programma sorgente* e viene tradotto (*compilato*) in un altro linguaggio (il *linguaggio oggetto*) quale l'Assembler o un linguaggio macchina, altrimenti segnala eventuali errori, grammaticali o logici, presenti nel testo. Sebbene il programma sorgente possa essere tradotto direttamente nel linguaggio oggetto, è utile codificare il programma sorgente in un metalinguaggio (*codice intermedio*) per due motivi:

- la traduzione del programma sorgente in un altro linguaggio oggetto è più agevole avendo già a disposizione il codice intermedio;
- la traduzione del programma sorgente può essere ottimizzata a prescindere dal linguaggio oggetto.

A tale scopo, la compilazione si svolge in due *fasi*: l'*analisi del testo* e la *sintesi del codice oggetto*. L'analisi svolge tre *funzioni*:

1) *analisi lessicale*
2) *analisi sintattica*
3) *analisi semantica*

e la sintesi altrettante:

4) *generazione del codice intermedio*
5) *ottimizzazione del codice intermedio*
6) *generazione del codice oggetto*

Mentre l'analisi dipende molto dal linguaggio sorgente ma è largamente indipendente dalla macchina, la sintesi è largamente indipendente dal linguaggio sorgente ma dipende molto dalla macchina. C'è da dire che l'interazione tra le diverse funzioni non è sequenziale, nel senso che la funzione i-esima può richiedere informazioni elaborate non solo dalla funzione $(i-1)$-esima ma anche da altre funzioni precedenti.
Oltre alla compilazione, altre funzioni possono concorrere a completare la traduzione del programma sorgente in un programma eseguibile. Ad esempio, se un programma contiene macroistruzioni oppure è suddiviso in moduli registrati in differenti file, il

compito di espandere le macroistruzioni e di mettere assieme i moduli è affidato ad un programma ad hoc, il *preprocessor*, che produrrà il programma sorgente da compilare. Inoltre, il codice oggetto prodotto dal compilatore può richiedere ulteriori modifiche perché possa essere mandato in esecuzione, ad esempio, può essere tradotto da un programma assembler in un *codice rilocabile* che verrà poi tradotto in un *codice macchina* da un programma di caricamento e da un programma di collegamento.

4.1 Le funzioni dell'analisi

4.1.1 *Analisi lessicale*

L'analisi lessicale ha due compiti principali: la *scansione* del testo, la sua *scomposizione* in lessemi e la sua *conversione* in una stringa di figure lessicali.

Con la scansione, il testo viene semplicemente letto carattere per carattere da sinistra a destra (o meglio dal primo all'ultimo carattere). Durante la scansione, vengono riconosciuti i separatori (e i commenti) che vengono eliminati dal testo; il testo così emendato è una stringa x che viene scomposta in lessemi, viene cioè segmentata in un certo numero di sottostringhe $x_1, \ldots, x_k$ di x tali che

— $x = x_1 \ldots x_k$

— per ogni h ($1 \leq h \leq k$), esiste una classe lessicale che contiene x_h.

Il risultato sarà la conversione di x in una stringa di figure lessicali

$$\mathbf{t}_1 \ldots \mathbf{t}_k$$

in cui $\mathbf{t}_h$ è la figura lessicale che dà il nome alla classe lessicale che contiene il lessema x_h ($1 \leq h \leq k$). Chiameremo tale stringa di figure lessicali lo *schema lessicale* del testo.

A titolo illustrativo, supponiamo che nel testo sia stato già isolato il lessema x_h quando, proseguendo nella scansione del testo, viene letta una lettera a. Se, continuando a scandire il testo, si arriva a leggere un carattere b che non è né una lettera né una cifra, allora tutta la sottostringa di x che inizia per a e termina con il carattere che precede b verrà riconosciuta come un lessema appartenente alla classe lessicale degli "identificatori". A questo punto, la scansione del testo può continuare a partire dal carattere b per individuare il lessema successivo.

Ogni volta che un lessema viene riconosciuto, il lessema viene registrato in un apposito catalogo (*tabella dei simboli*).

Consideriamo ad esempio un testo che contenga la seguente istruzione nel linguaggio di programmazione Pascal:

```
costo := canone + consumo * 10      (1)
```

L'analisi lessicale del testo porta a riconoscere i seguenti lessemi:

`costo`	è un valore della figura lessicale **id**
`:=`	è il valore della figura lessicale **ass**
`canone`	è un valore della figura lessicale **id**
`+`	è un valore della figura lessicale **add**
`consumo`	è un valore della figura lessicale **id**
`*`	è un valore della figura lessicale **mol**
`10`	è un valore della figura lessicale **num**

e, così, viene costruito lo schema lessicale del testo (1)

id ass id add id mol num (2)

Si osservi che in questa stringa non c'è traccia dei singoli identificatori (`costo`, `canone` e `consumo`) né degli operatori aritmetici di addizione e moltiplicazione. D'altra parte, la traduzione del testo nel codice oggetto non ne potrà prescindere. A questo scopo, i tre identificatori vengono registrati nella tabella dei simboli. Si vedranno pù in là i dettagli; per il momento, aggiungiamo semplicemente a ciascuna occorrenza delle figure lessicali **id** e **num** nello schema lessicale (2) un indice che ne distingue il valore, ed esplicitiamo i valori delle figure lessicali **ass**, **add** e **mol**. Così, lo schema lessicale (2) diventa

$\mathbf{id}_1$ **:=** $\mathbf{id}_2$ **+** $\mathbf{id}_3$ ***** $\mathbf{num}_1$ (3)

che chiamiamo lo *schema lessicale annotato* del testo (1).

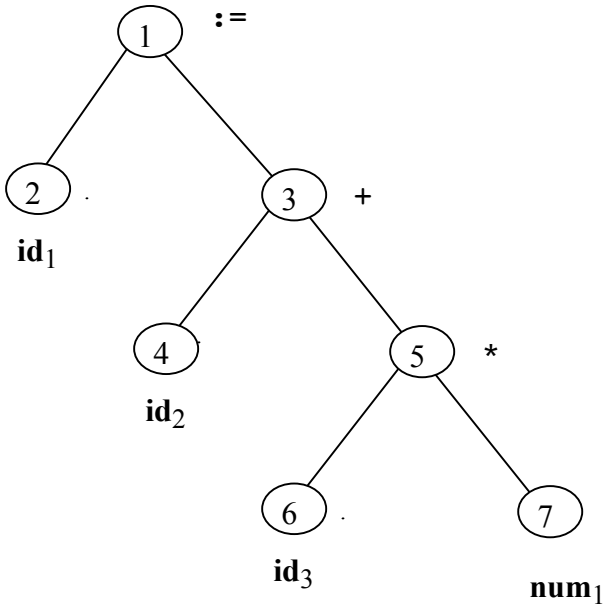
4.1.2 *Analisi sintattica*

Nell'analisi sintattica lo schema lessicale del testo viene analizzato per decidere se esso è o meno una frase della grammatica ***G*** che definisce la sintassi del linguaggio sorgente. Se questo è il caso, viene costruito l'albero di derivazione dello schema lessicale del programma sorgente.

4.1.3 *Analisi semantica*

Nell'analisi semantica, entra in gioco la componente semantica del linguaggio sorgente: ad ogni simbolo grammaticale è associato un insieme di *attributi* e, per ogni produzione, è specificato un insieme di *regole semantiche* per il calcolo degli attributi associati ai simboli che compaiono nella produzione. Sulla base delle specifiche semantiche delle produzioni della grammatica ***G***, dello schema lessicale annotato del programma sorgente e dell'albero di derivazione per il suo schema lessicale, viene così costruito un modello operazionale del programma sorgente che evidenzia gli

operatori e gli operandi delle espressioni ed istruzioni. Il modello operazionale è ancora in forma di albero (l'*albero della sintassi* del programma sorgente).

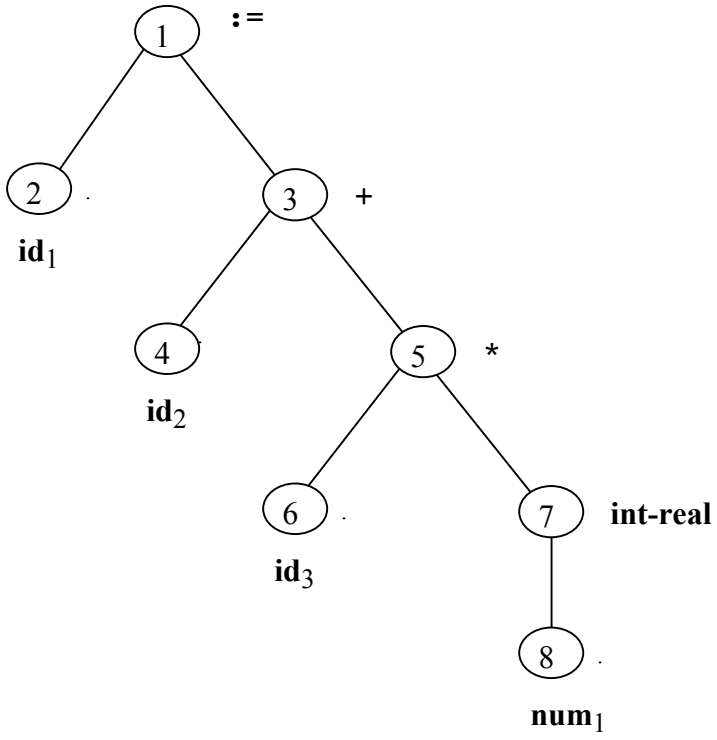


Si possono costruire alberi sintattici per qualsiasi costrutto sintattico. Per esempio, per il costrutto

while (*expr*) *stmt*

del linguaggio C, dove *expr* e *stmt* stanno per espressione e istruzione, avremo un nodo operatore **while** con due figli *expr* e *stmt*.

A partire dall'albero sintattico, vengono effettuati controlli per accertarsi che le parti del programma sorgente si integrino senza creare problemi di correttezza sotto il profilo semantico. A questo punto, viene eseguito il *controllo dei tipi*: ogni operatore deve avere operandi del tipo permesso dal linguaggio sorgente. Comunque, la specifica semantica del linguaggio sorgente può permettere di forzare un tipo. Ad esempio, supponiamo che il testo che contiene l'istruzione (1) specifichi in qualche altra sua parte che tutti gli identificatori siano di tipo reale e che il numero `10` (il valore di **num**) sia riconosciuto come intero. Allora, il numero `10` verrà convertito nel corrispondente numero reale (`10.0`). A tale scopo, nell'albero della sintassi viene inserito un nodo aggiuntivo etichettato con l'operatore **int-real** che trasforma `10` in `10.0`.



4.2 Le funzioni della sintesi

4.2.1 *Generazione del codice intermedio*

A partire dall'albero della sintassi del programma sorgente, il compilatore genera un *codice intermedio*, che può pensarsi come un programma eseguibile su di una macchina "astratta". Il codice intermedio si ottiene "linearizzando" l'albero della sintassi dello schema lessicale e associando un nome proprio ad ogni nodo interno. Tipicamente, il codice intermedio è una lista di istruzioni, ciascuna della cosiddetta forma *a tre indirizzi*, una per ogni nodo operatore dell'albero della sintassi. Ad esempio, la codifica a tre indirizzi di (3) potrebbe essere

```
temp1 := int-real(10)
temp2 := id3 * temp1                    (4)
temp3 := id2 + temp2
id1 := temp3
```

Qui, ogni istruzione ha al più un operatore in aggiunta all'operatore di assegnazione; va osservato che il compilatore deve aver già determinato l'ordine di precedenza in cui vanno eseguito le operazioni aritmetiche.

Una tipica istruzione del codice intermedio è dunque della forma

```
x := y op z
```

dove `x`, `y` e `z` sono nomi, costanti o variabili temporanee generate dal compilatore, e *op* è un generico operatore aritmetico o un operatore logico su grandezze booleane. Così, un'espressione del programma sorgente del tipo

```
x + y * z
```

viene codificata come

```
t1 := y * z
t2 := x + t1
```

dove t_1 e t_2 sono variabili temporanee.
La forma a tre indirizzi per le istruzioni del metalinguaggio è scelta per la facilità sia della generazione del codice oggetto che dell'ottimizzazione del codice intermedio. Qui di seguito sono riportate altre istruzioni a tre indirizzi che sono più comunemente usate nei metacodici:

1. `x := y` *op* `z` — dove *op* è un operatore diadico
2. `x :=` *op* `y` — dove *op* è un operatore monadico
3. `x := y`
4. `goto` *e* — dove *e* è l'etichetta di un'altra istruzione a tre indirizzi
5. `if x` *comp* `y goto` *e*
6. `x := y[i]`

Così, il codice intermedio somiglia ad un programma scritto in linguaggio assemblativo per una macchina in cui ogni cella di memoria può fungere da registro.

4.2.2 *Ottimizzazione del codice intermedio*
Il codice intermedio è la traduzione dell'albero della sintassi ottenuta nell'analisi semantica. Un programma più efficiente di (4) consterebbe semplicemente delle due istruzioni

```
temp1 := id3 * 10.0
id1 := id2 + temp1
```

4.2.3 *Generazione del codice oggetto*
L'ultima funzione è la generazione del codice oggetto, che normalmente è un codice macchina rilocabile oppure un codice assemblativo. Se il codice oggetto è scritto in

linguaggio assemblativo, esso sarà poi passato ad un programma assembler per la generazione di un codice macchina rilocabile.
Per generare il codice oggetto, ogni istruzione del codice intermedio ottimizzato è tradotta in una sequenza di istruzioni macchina. Un aspetto cruciale di tale traduzione è l'assegnazione delle variabili ai registri. Per esempio, un codice oggetto potrebbe essere

```
MOVF      id3, R2
MULTF     #10.0, R2
MOVF      id2, R1
ADDF      R2, R1
MOVF      R1, id1
```

Qui, il primo e il secondo operando di ogni istruzione specificano rispettivamnete l'origine e la destinazione. Inoltre, il suffisso `F` nelle istruzioni specifica che le operazioni trattano numeri in virgola mobile. Infine, il prefisso # di `#10.0` specifica che il numero `10.0` è trattato come una costante numerica.

4.3 Passaggi

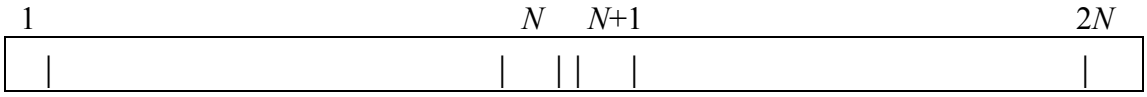
Nella sua implementazione, il processo di compilazione si svolge in uno o più *passaggi*. Ogni passaggio consiste nella lettura di un file di ingresso e nella scrittura di un file di uscita. Diverse funzioni della compilazione possono essere svolte in un unico passaggio. Ora, se è conveniente avere pochi passaggi perché leggere e scrivere un file ha sempre un costo, è altrettanto vero che implementare l'intera compilazione in un unico passaggio costringerebbe a tenere in memoria l'intero programma sorgente ma questo non è conveniente dal momento che, come si è detto, l'interazione tra le diverse funzioni non è sequenziale. Inoltre, le rappresentazioni intermedie del programma sorgente (l'albero di derivazione, l'albero di derivazione annotato, l'albero della sintassi, il codice intermedio) possono avere dimensioni molto maggiori sia di quelle del programma sorgente sia di quelle del codice oggetto. Comunque per alcune funzioni, come quelle dell'analisi, accorparle in un unico passaggio non crea grossi problemi. In tal caso, a "condurre il gioco" è il programma che svolge l'analisi sintattica (*parser*). Di volta in volta, il parser ordina al programma che svolge l'analisi lessicale di riconoscere ed inviargli un lessema che il parser andrà poi ad integrare nell'albero della sintassi già costruito dopodiché il parser ordina al generatore del codice oggetto di sviluppare la porzione del codice già ottenuto.

Cap. 5

ANALISI LESSICALE

5.1 Scansione

La lettura di un carattere del testo viene fatta con l'ausilio di un *buffer di ingresso*. Dal momento che la lettura può richiedere molto tempo, sono stati studiati alcuni schemi di *buffering* per ridurre i costi computazionali. Usualmente si fa uso di un *buffer doppio*, che consta di due parti uguali, che possono contenere ciascuna N caratteri, se N è la capacità di un blocco di disco (ad esempio, $N = 1024$ o $N = 4096$).



Buffer di ingresso

Con un singolo comando di lettura, il contenuto di un intero blocco (piuttosto che un singolo carattere) viene copiato in una delle due metà del buffer; se la parte del testo ancora da leggere contiene meno di N caratteri, allora il blocco sarà solo parzialmente occupato e un carattere speciale `$` (che segnala la fine del testo) viene aggiunto dopo l'ultimo carattere del testo.

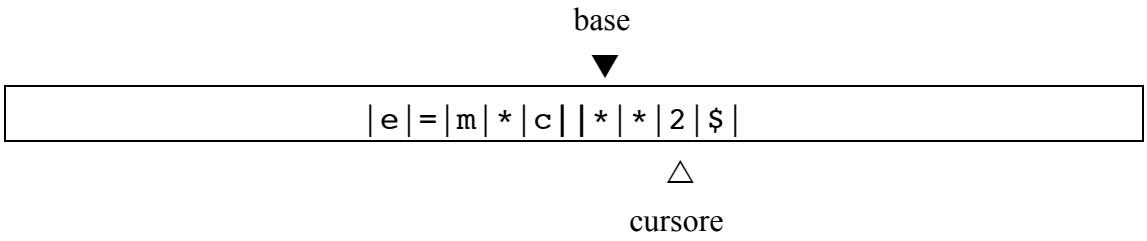
Vengono poi usati due puntatori, che chiamiamo la *base* e il *cursore*. Inizialmente, entrambi i puntatori indicano la posizione (nel buffer) del primo carattere da cui inizia la ricerca del lessema da isolare. Ma, mentre la base viene tenuta ferma, il cursore viene incrementato man mano che un nuovo carattere viene letto finché non viene riconosciuto un lessema che si estende dalla posizione indicata dalla base fino ad una certa posizione che precede quella indicata dal cursore. Effettuato il riconoscimento, al cursore verrà assegnata la posizione successiva a quella dell'ultimo carattere del lessema riconosciuto; quindi, dopo l'elaborazione del lessema, alla base verrà assegnata la stessa posizione che indica il cursore. Se il cursore è in prossimità della fine di una delle due metà del buffer, l'altra metà verrà riempita di N caratteri cosicché, quando il cursore viene fatto avanzare, un qualche carattere di ingresso sarà pronto per essere letto. Nel caso della seconda metà, l'avanzamento del cursore consisterà semplicemente nell'assegnargli la prima posizione del buffer. L'avanzamento del cursore è schematicamente riassunto nella seguente procedura:

```
if cursore = N then
        begin
        caricare la seconda metà;
        cursore := cursore + 1
        end
else    if cursore = 2N then
                begin
                caricare la prima metà;
                cursore := 1;
                end
        else cursore := cursore + 1;
```

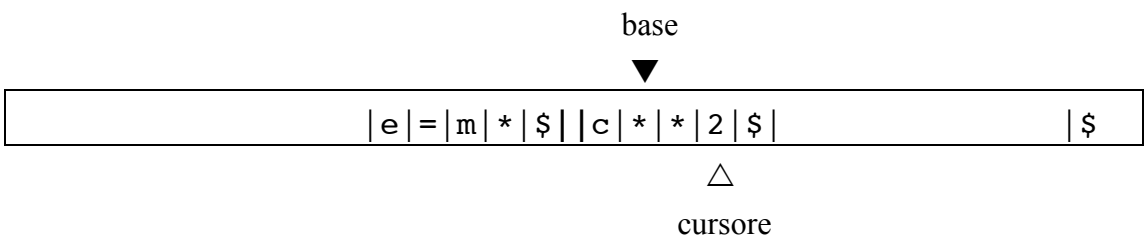
Esempio 5.1 Consideriamo un programma in Fortran che termina con l'istruzione

```
e=m*c**2
```

Dopo aver riconosciuto il lessema `c` ed aver letto i due asterischi, la situazione è la seguente:



Fatta eccezione per i due casi in cui il cursore si trovi alla fine di una delle due metà del buffer, ogni suo avanzamento richiede l'esecuzione di due test. Per dimezzare il numero di test, conviene introdurre un carattere speciale, che chiamiamo *sentinella*, che serve a segnalare la fine di una metà del buffer. Una scelta ovvia per il carattere sentinella è lo stesso segno `$` di fine testo.



Con questo espediente la procedura di avanzamento del cursore diventa

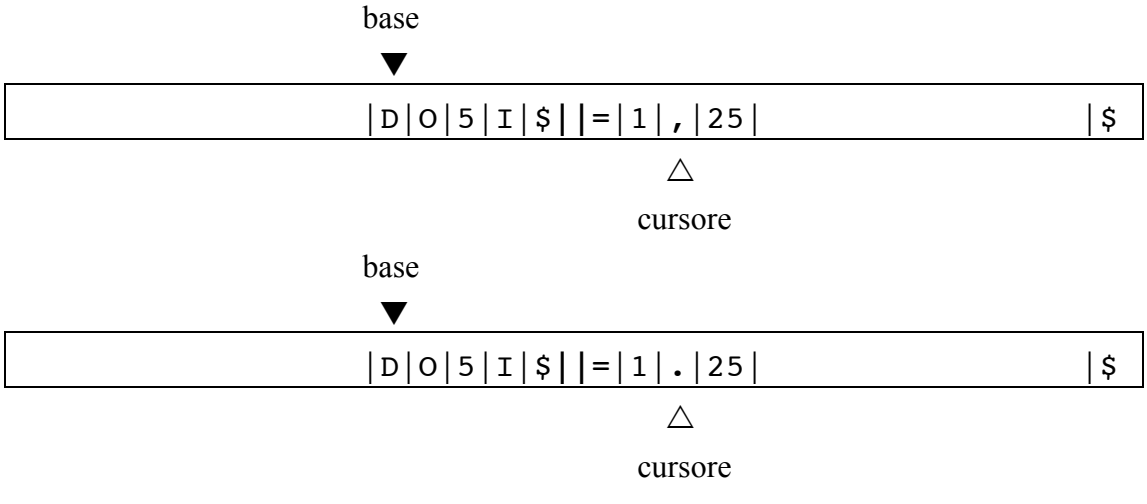
```
cursore := cursore + 1;
if cursore↑ = $ then
        begin
        if cursore = N then
                begin
                caricare la seconda metà;
                cursore := cursore + 1
                end

        else    if cursore = 2N then
                        begin
                        caricare la prima metà;
                        cursore := 1;
                        end
                else    /* in tal caso $ indica fine del testo */  Esci
        end
```

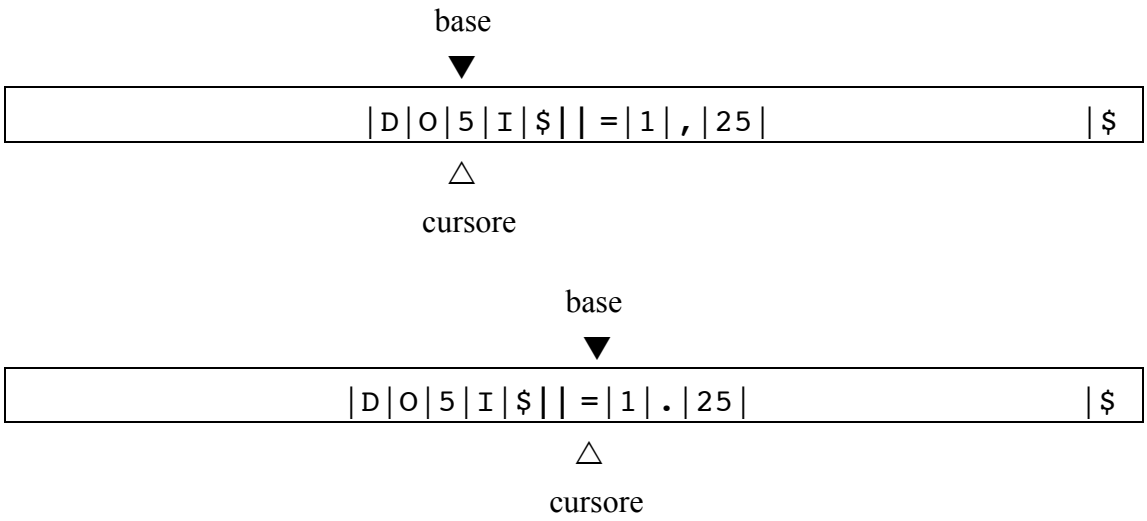
Spesso, prima che un lessema venga riconosciuto, il cursore dovrà avanzare ben oltre la posizione dell'ultimo carattere del lessema. Il numero di avanzamenti del cursore, che chiamiamo la *corsa* del cursore, può variare molto specialmente quando le parole chiave non hanno un uso esclusivo nel linguaggio sorgente. Per esempio, per le due stringhe contenute in un programma scritto in Fortran

```
DO5I=1,25                    DO5I=1.25
```

la scansione avanzerà fino al carattere successivo a `1`.



Per la prima stringa, è la sequenza `DO` ad essere riconosciuta come lessema (una parola chiave); per la seconda, è la sequenza `DO5I` ad essere riconosciuta come lessema (un identificatore). Effettuato il riconoscimento, inizierà la ricerca del lessema successivo con i due puntatori nelle posizioni indicate qui di seguito



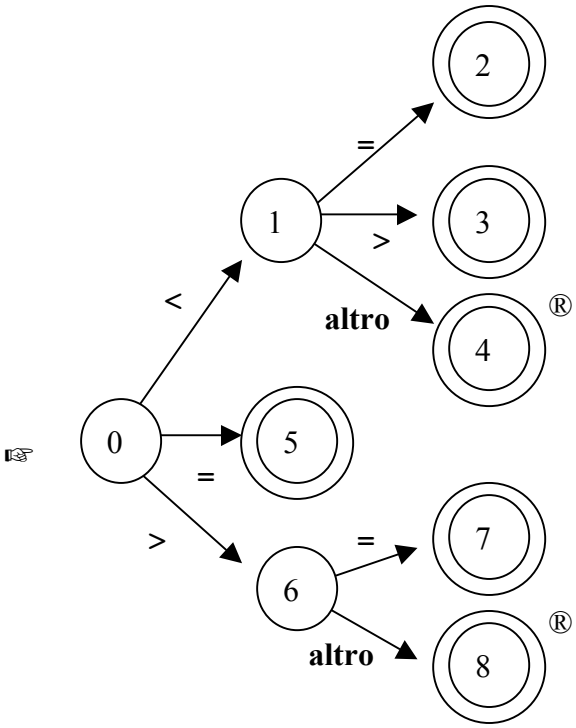
Lo stesso avviene quando si incontra il segno `<` come primo carattere. Se il carattere successivo è `=`, allora è la sequenza `<=` ad essere riconosciuta come lessema; altrimenti, è il carattere `<` ad essere riconosciuto come lessema.

5.2 Riconoscimento dei lessemi

Per il riconoscimento dei lessemi potremmo far uso delle grammatiche lessicali e, siccome queste sono grammatiche regolari, potremmo applicare l'algoritmo CYK per il riconoscimento. Questa procedura è però onerosa sicché, sia per la definizione delle classi lessicali sia per il riconoscimento dei lessemi, si preferisce far ricorso ad un formalismo che (come si vedrà nei prossimi capitoli) è del tutto equivalente a quello delle grammatiche regolari. Ogni classe lessicale è definita da un *automa finito* e, durante la scansione, i diversi automi finiti sono esaminati secondo un ordine che dà la precedenza a quelli che corrispondono alle classi lessicali che capitano più frequentemente, visto che si accederà al diagramma delle transizioni di un automa solo dopo che la ricerca con i precedenti è stata infruttuosa. A tale scopo gli "stati iniziali" degli automi sono numerati in maniera tale che, se un automa precede un altro, allora lo stato iniziale del primo ha una numerazione inferiore allo stato iniziale del secondo. Supponiamo che, quando il cursore indica un certo carattere a, l'automa in uso è $\boldsymbol{A}$ che si trova nello stato q. Se $\boldsymbol{A}$ non ha nessuna transizione su a con stato iniziale q, allora il cursore viene retrocesso e si continua dallo stato iniziale dell'automa successivo ad $\boldsymbol{A}$. Se invece esiste una transizione su a con stato iniziale q, allora la scansione può procedere a meno che lo stato finale della transizione sia uno stato di accettazione di $\boldsymbol{A}$; in tal caso, viene univocamente determinato un lessema appartenente alla classe lessicale corrispondente ad $\boldsymbol{A}$. Il riconoscimento del lessema

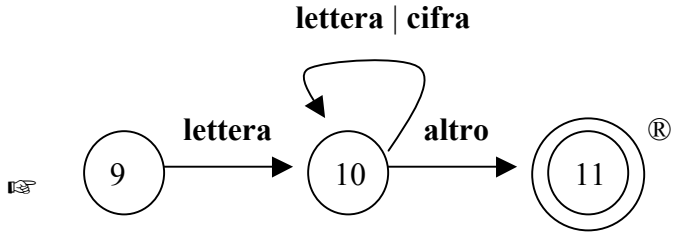
termina con l'esecuzione eventuale di un comando per far retrocedere (se necessario) il cursore del buffer d'ingresso.
Diamo alcuni esempi di automi finiti associati a classi lessicali.

— *Operatori comparativi*. Per il riconoscimento degli operatori comparativi abbiamo un solo diagramma delle transizioni che ha lo stato 0 come stato iniziale



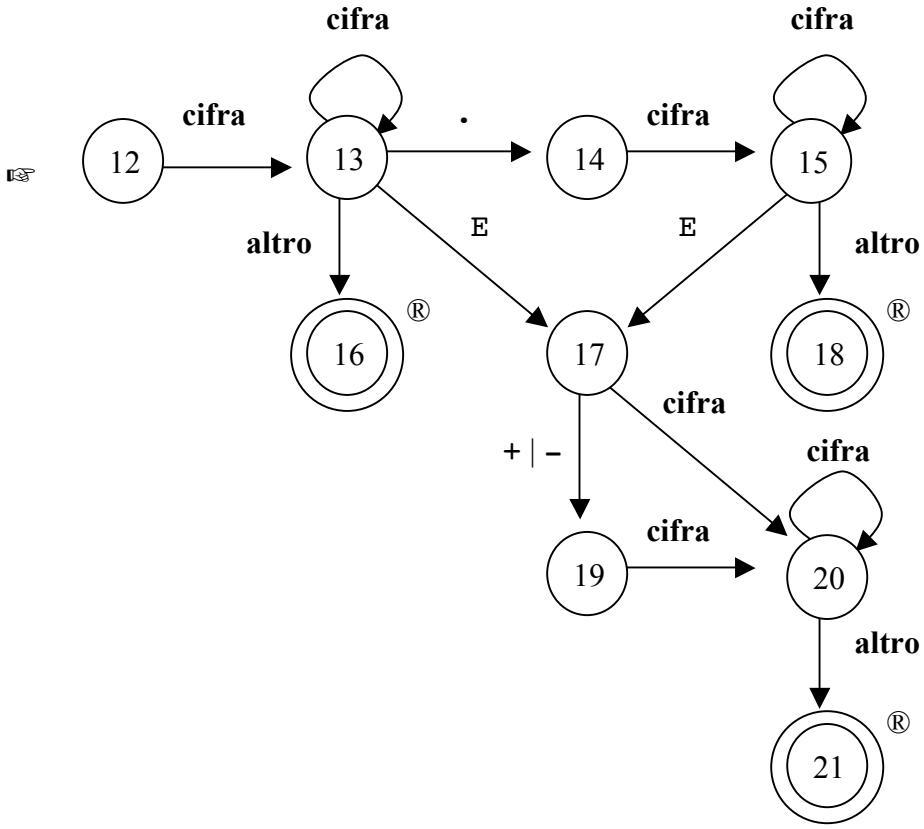
e gli stati 2, 3, 4, 5, 7 e 8. Gli stati 2, 3, 5 e 7 riconoscono rispettivamente i lessemi <=, <>, = e >=; gli stati 4 e 8 riconoscono rispettivamente i lessemi < e >, inoltre, essi prevedono la retrocessione del cursore.

— *Identificatori e parole chiave*. Per il riconoscimento sia delle parole chiave che degli identificatori abbiamo un solo diagramma delle transizioni che ha lo stato 9 come stato iniziale

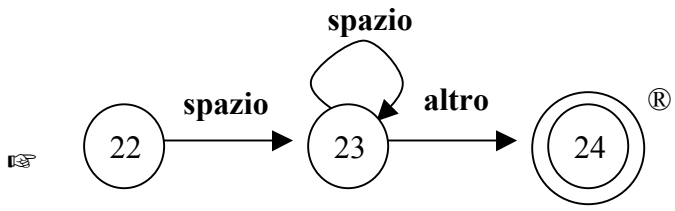


e lo stato 11 come stato di accettazione. Lo stato 11 riconosce i lessemi formati dalla concatenazione dei caratteri che hanno etichettano tutte le transizioni dallo stato 9 allo stato 11 escluso; inoltre, lo stato 11 prevede la retrocessione del cursore.

— *Numeri assoluti*. Per il riconoscimento dei numeri senza segno facciamo uso del seguente diagramma delle transizioni che ha come stato iniziale lo stato 12 e come stati di accettazione gli stati 16, 18 e 21 (tutti e tre con azione di retrocessione del cursore).



— *Separatori*. Per i separatori abbiamo un unico diagramma delle transizioni che ha come stato iniziale lo stato 22 e come stato di accettazione lo stato 24 (con azione di retrocessione del cursore).



5.3 Risultato dell'analisi lessicale

La lista dei diagrammi delle transizioni viene tradotta in un programma le cui dimensioni sono proporzionali al numero totale degli stati e delle transizioni dei diagrammi. Ogni stato viene tradotto in un segmento di codice.

Cominciamo dagli stati da cui escono delle transizioni e sia *q* uno di questi. Il segmento di programma corrispondente a *q* attiva una routine che

— legge un carattere dal buffer d'ingresso (quello che occupa la posizione indicata dal cursore),

— fa avanzare il cursore e

— restituisce il carattere letto.

A questo punto, se esiste una transizione con stato iniziale *q* che corrisponde al carattere letto, il controllo viene passato allo stato finale della transizione; altrimenti, viene attivata una routine che

— fa retrocedere il cursore riportandolo nella posizione della base e

— passa il controllo allo stato iniziale del successivo diagramma di transizione.

Quanto poi agli stati di accettazione, a ciascuno è associata una *procedura di identificazione* che fornisce non solo il lessema trovato e la figura lessicale **t** di cui il lessema è un valore ma anche altri *attributi* del lessema necessari alla compilazione: tipo, campo d'azione, etc. È nella *tabella dei simboli* che troveremo riportati i lessemi con i loro attributi, una riga della tabella per ogni lessema.
Supponiamo che sia stato riconosciuto un lessema *x* in corrispondenza dello stato di accettazione *q*. A titolo illustrativo, diamo alcuni esempi a seconda dello stato *q*.

— Se *q* = 2, 3, 4, 5, 7 o 8 allora la procedura di identificazione restituisce rispettivamente le coppie:

(**comp**: `LE`)
(**comp**: `NE`)
(**comp**: `LT`)
(**comp**: `EQ`)
(**comp**: `GE`)
(**comp**: `GT`).

— Se *q* = 11, allora la procedura di identificazione va a consultare la tabella dei simboli, che contiene una riga per ogni lessema che sia una parola chiave o un identificatore. La tabella dei simboli è un array di record con tre o più campi: *lessema*, *figura*, *tipo* … Ad esempio, supponiamo che le parole chiave del linguaggio sorgente siano

`program`

```
var
array
of
integer
real
function
procedure
begin
end
if
then
else
while
do
const
div
mod
and
```

La tabella dei simboli è inizializzata riservando le righe da 1 a 19 alle parole chiave mentre la riga 0 è vuota.

	lessema	*figura*	*tipo*
0			
1	`program`	**pgm**	-
	...	…	…
19	`and`	**and**	-

Tabella dei simboli

Quando viene riconosciuto il lessema *x*, questo viene cercato nella tabella dei simboli. Se la ricerca dà esito positivo ed *x* è il lessema corrispondente alla riga *n*-esima (nella fattispecie, se *x* = `program` allora *n* = 1 e se *x* = `and` allora *n* = 19), allora il valore del campo *figura* nella riga *n*-esima è uguale a **t** e la procedura di identificazione restituisce la coppia (**t**: *x*) o (**t**: *n*) a seconda che si tratti di una parola chiave oppure di un identificatore. Se invece la ricerca dà esito negativo, allora il lessema *x* viene riconosciuto come un identificatore; a questo punto, la procedura di identificazione provvede ad aggiungere alla tabella dei simboli una nuova riga (*lessema* = *x*, *figura* = **id**, …) nella posizione ventesima e fornisce come risultato (**id**: 20) cosicché *x* = 20↑*.lessema*.

	lessema	*figura*	*tipo*
0			
1	`program`	**pgm**	-
	…	**…**	…
19	`and`	**and**	-
20	*x*	**id**	-

Tabella dei simboli

— Se q = 16, 18, 21 allora la procedura di identificazione dà come risultato (**num**: n) dove n è il numero di riga nella tabella dei simboli corrispondente al lessema numerico x. Nel seguito, per semplicità, talora assumeremo che la procedura di identificazione restituisca sempre la coppia (**num**: x).

— Se q = 24, allora il lessema viene ignorato.

Esempio 5.1 (seguito) Riprendiamo l'istruzione `e:=m*c**2`

	lessema	*figura*	*tipo*
1	`program`	**pgm**	-
	…	**…**	…
19	`and`	**and**	-
20	`e`	**id**	-
21	`m`	**id**	-
22	`c`	**id**	-
23	`2`	**num**	

Tabella dei simboli

L'analisi lessicale della sequenza `e:=m*c**2` porta a riconoscere i seguenti lessemi:

`e`	(**id**: 20)
`:=`	(**ass**)
`m`	(**id**: 21)
`*`	(**mol**: `*`)
`c`	(**id**: 22)
`**`	(**exp**)
`2`	(**num**: 23)

Così, lo schema lessicale dell'istruzione `e:=m*c**2` è

id ass id mol id exp num

e il suo schema lessicale annotato è

(**id**: 20) **ass** (**id**: 21) (**mol**: `*`) (**id**: 22) **exp** (**num**: 23)

Cap. 6

AUTOMI FINITI

Come anticipato nel capitolo precdente, le classi lessicali sono specificate con "automi finiti", il cui formalismo ha lo stesso potere espressivo delle grammatiche regolari. Il vantaggio che se ne trae è nella facilità con cui viene risolto il problema del riconoscimento dei lessemi.

Un automa finito ha un insieme finito di stati e può muoversi da uno stato all'altro in risposta ad un input esterno (la lettura di un carattere). Una distinzione fondamentale che va fatta tra automi finiti attiene al "determinismo" della risposta: in ogni istante un automa finito deterministico può trovarsi in un solo stato, mentre un automa finito nondeterministico può trovarsi in più stati. Sorprendentemente, il nondeterminismo non aggiunge espressività al formalismo nel senso che ogni linguaggio definito da un automa finito nondeterministico può essere definito per mezzo di un automa finito deterministico. A suo merito però, il nondeterminismo ha il vantaggio che spesso un linguaggio può essere definito in maniera più semplice.

6.1 Automi finiti deterministici

Un *automa finito deterministico* (AFD) è definito da una quintupla $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$ dove

> Q è un insieme finito i cui elementi sono chiamati gli *stati* di $\boldsymbol{A}$;
>
> Σ è un alfabeto;
>
> δ (*funzione di transizione* di $\boldsymbol{A}$) è una funzione definita su $Q \times \Sigma$ con codominio Q;
>
> q_0 è uno stato particolare, chiamato *stato iniziale* (o *di partenza*) di $\boldsymbol{A}$;
>
> F è un sottoinsieme di Q, i cui elementi sono detti *stati finali* (o *di accettazione*) di $\boldsymbol{A}$.

Vediamo come l'ADF fa a decidere se "accettare" o meno una stringa di caratteri $a_1 \ldots a_n$ su Σ. Siano $q_1 = \delta(q_0, a_1)$, $q_2 = \delta(q_1, a_2)$, …, $q_n = \delta(q_{n-1}, a_n)$. Inizialmente, $\boldsymbol{A}$ è nel suo stato iniziale q_0. Successivamente, quando viene letto l'i.mo simbolo della stringa in esame, $i = 1, \ldots, n$, l'AFD si porta nello stato q_i. Infine, se q_n appartiene ad F (è cioè uno stato di accettazione), allora la stringa $a_1 \ldots a_n$ è "accettata" da $\boldsymbol{A}$, altrimenti essa è "rifiutata". Per avere una definizione più precisa delle stringhe accettate da $\boldsymbol{A}$, introduciamo la nozione di *funzione di transizione estesa* $\hat{\delta}$: questa è una funzione con dominio $Q \times \Sigma^*$ e codominio Q che è definita in maniera induttiva sulla lunghezza della stringa su Σ:

per ogni stato q di $\boldsymbol{A}$ $\quad\quad \hat{\delta}(q, \varepsilon) = q$

per ogni stato q di $\boldsymbol{A}$, per ogni stringa y e per ogni carattere a

$$\hat{\delta}(q, ya) = \delta(\hat{\delta}(q, y), a)$$

Si osservi che si ha sempre $\hat{\delta}(q, a) = \hat{\delta}(q, \varepsilon a) = \delta(\hat{\delta}(q, \varepsilon), a) = \delta(q, a)$.

Una stringa x su Σ è *accettata* da $\boldsymbol{A}$ se e solo se $\hat{\delta}(q_0, x)$ è uno stato di accettazione di $\boldsymbol{A}$. L'insieme di siffatte stringhe prende il nome di *linguaggio accettato* da $\boldsymbol{A}$, che indichiamo con $L(\boldsymbol{A})$:

$$L(\boldsymbol{A}) = \{x \in \Sigma^* : \hat{\delta}(q_0, x) \in F\}.$$

Dunque, il riconoscimento delle stringhe appartenenti ad $L(\boldsymbol{A})$ richiede il calcolo dello stato $\hat{\delta}(q_0, x)$ se x è la stringa in esame, cosa che richiede un tempo $O(|x|)$ che, si noti, è indipendente dal numero $|Q|$ degli stati di $\boldsymbol{A}$.

Per comodità, un AFD viene comunemente specificato mediante una "tabella delle transizioni" oppure mediante un "diagramma delle transizioni", dove per *transizione* si intende una terna (q, a, q') tale che $q' = \delta(q, a)$; inoltre, gli stati q e q' sono detti rispettivamente lo *stato iniziale* e lo *stato finale* della transizione (q, a, q').
La *tabella delle transizioni* di $\boldsymbol{A}$ è la tabella le cui righe sono indicizzate da Q e le cui colonne sono indicizzate da Σ; inoltre, la generica cella (q, a) della tabella riporta lo stato $\delta(q, a)$, cioè lo stato finale della transizione di $\boldsymbol{A}$ su a che stato iniziale q.
Il *diagramma delle transizioni* di $\boldsymbol{A}$ è un grafo orientato con insieme di vertici Q (v. Appendice), in cui ogni arco (q, q') è etichettato con il sottoinsieme di Σ formato da quei caratteri a per cui (q, a, q') è una transizione di $\boldsymbol{A}$. Chiamiamo infine *dimensione* di $\boldsymbol{A}$ il numero dei vertici più la somma delle cardinalità degli insiemi che etichettano gli archi del diagramma delle transizioni di $\boldsymbol{A}$. Così, la dimensione di $\boldsymbol{A}$ è $O(|Q| \cdot |\Sigma|)$.

Esempio 6.1 Consideriamo l'AFD $\boldsymbol{A}$ con $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, la cui tabella delle transizioni è mostrata in figura

q	a	b
→ 0	1	2
1	1	3
2	1	2
3	1	4
○ 4	1	2

Tabella delle transizioni di $\boldsymbol{A}$

Il diagramma delle transizioni di ***A*** è mostrato in Figura.

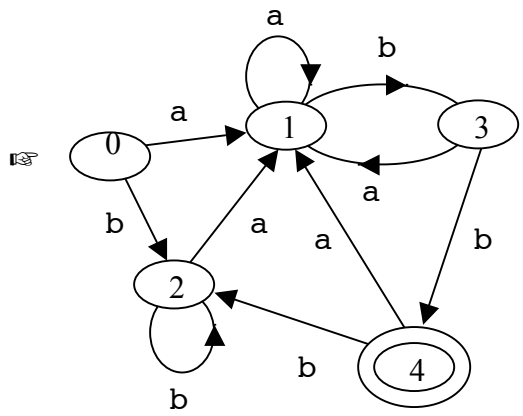


Diagramma delle transizioni di ***A*** ■

Come accennato all'inizio di questo capitolo, il formalismo degli automi finiti ha lo stesso potere espressivo delle grammatiche regolari. A questo punto, possiamo già provare che il formalismo degli AFD non è più espressivo di quello delle grammatiche regolari, e rimandiamo al Teorema 5.1 la completa dimostrazione della equivalenza.

Lemma 6.1 Ogni linguaggio accettato da un automa finito deterministico è regolare.
Dimostrazione. Dato un AFD $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$, si consideri la grammatica regolare $\boldsymbol{G} = (N, T, P, S)$ così definita:

$$N = Q$$
$$T = \Sigma$$
$$P = P_1 \cup P_2$$
$$S = q_0$$

dove

$$P_1 = \{q \rightarrow aq' : q' = \delta(q, a)\} \qquad P_2 = \{q \rightarrow \varepsilon : q \in F\}.$$

Si osservi che le produzioni in P_1 sono tutte monotone crescenti. Consideriamo ora una stringa x di lunghezza k. Distinguiamo due casi a seconda che $k = 0$ o $k > 0$.

Caso 1: $k = 0$, cioè x è la stringa vuota. Lo stato finale di ***A*** con input x è lo stato q_0. Ne segue che ***A*** accetta x se e solo se q_0 appartiene ad F. D'altra parte, siccome le produzioni in P_1 sono tutte monotone crescenti, x è derivabile dal simbolo iniziale S di ***G*** e, quindi x è una frase di ***G***, se e solo se P_2 contiene la produzione $S \rightarrow \varepsilon$, cioè se e solo se $q_0 \in F$. Dunque, ***A*** accetta x se e solo se x è una frase di ***G***.

Caso 2: $k > 0$. Sia $x = a_1 \ldots a_k$ e sia $(q_0, q_1, \ldots, q_k)$ la sequenza degli stati di ***A*** con input x. Ne segue che ***A*** accetta x se e solo se q_k appartiene ad F. D'altra parte, $a_1 a_2 \ldots$

$a_k q_k$ è l'unica formula di $\boldsymbol{G}$ di lunghezza k+1 che abbia x come prefisso e che sia derivabile dal simbolo iniziale S di $\boldsymbol{G}$ applicando solo produzioni in P_1:

$(q_0, a_1 q_1, a_1 a_2 q_2, \ldots, a_1 a_2 \ldots a_k q_k)$.

Ne segue che x è una frase di $\boldsymbol{G}$ se e solo se P_2 contiene la produzione $q_k \rightarrow \varepsilon$, cioè se e solo se $q_k \in F$. Dunque, $\boldsymbol{A}$ accetta x se e solo se x è una frase di $\boldsymbol{G}$.

Abbiamo così provato che $L(\boldsymbol{A}) = L(\boldsymbol{G})$. □

Esempio 6.1 (seguito) Per il Lemma 6.1, il linguaggio accettato da $\boldsymbol{A}$ coincide con il linguaggio generato dalla grammatica regolare $\boldsymbol{G} = (N, T, P, 0)$ così definita:

$N = \{0, 1, 2, 3, 4\}$

$T = \{\mathtt{a}, \mathtt{b}\}$

e P contiene le seguenti undici produzioni:

$0 \rightarrow \mathtt{a1} \mid \mathtt{b2}$
$1 \rightarrow \mathtt{a1} \mid \mathtt{b3}$
$2 \rightarrow \mathtt{a1} \mid \mathtt{b2}$
$3 \rightarrow \mathtt{a1} \mid \mathtt{b4}$
$4 \rightarrow \mathtt{a1} \mid \mathtt{b2} \mid \varepsilon$ ■

Nei prossimi due paragrafi tratteremo il problema dell'equivalenza tra due AFD e il problema di trovare un AFD equivalente ad uno dato e di dimensione minima (cioè con il minor numero di stati).

6.1.1 *Equivalenza tra automi finiti deterministici*
Due AFD $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$ e $\boldsymbol{A}' = (Q', \Sigma, \delta', q'_0, F')$ sono *equivalenti* se $L(\boldsymbol{A}) = L(\boldsymbol{A}')$. Per decidere se $\boldsymbol{A}$ e $\boldsymbol{A}'$ sono equivalenti, possiamo far ricorso all'*algoritmo di Moore* che presentiamo nel caso che $\Sigma = \{a, b\}$. L'algoritmo fa uso di una tabella (la *tabella di Moore*), inizialmente vuota, con 3 (= $|\Sigma|$ + 1) colonne in cui ogni riga è destinata a contenere tre coppie di stati del tipo:

$\langle q, q' \rangle \qquad \langle \delta(q, a), \delta'(q', a) \rangle \qquad \langle \delta(q, b), \delta'(q', b) \rangle$

con $\langle q, q' \rangle \in Q \times Q'$. Il primo passo consiste nell'esame della coppia $\langle q_0, q'_0 \rangle$; l'esame dà esito positivo se e solo se

$q_0 \in F$ e $q'_0 \in F'$ oppure $q_0 \notin F$ e $q'_0 \notin F'$.

Se l'esame dà esito negativo (cioè, $q_0 \in F$ e $q'_0 \notin F'$ oppure $q_0 \notin F$ e $q'_0 \in F'$), si conclude subito che $\boldsymbol{A}$ e $\boldsymbol{A}'$ non sono equivalenti. Altrimenti (cioè se l'esame dà esito

positivo), la coppia $\langle q_0, q'_0 \rangle$ viene posta nella cella della prima riga corrispondente alla prima colonna. A questo punto, la prima riga della tabella di Moore viene "completata" calcolando le due coppie di stati

$$\langle \delta(q_0, a), \delta'(q'_0, a) \rangle \quad \langle \delta(q_0, b), \delta'(q'_0, b) \rangle$$

A questo punto, è la coppia $\langle \delta(q_0, a), \delta'(q'_0, a) \rangle$ ad essere esaminata a meno che non sia stata già presa in esame, cioè a meno che $\delta(q_0, a) = q_0$ e $\delta'(q'_0, a) = q'_0$. Se l'esame dà esito positivo, allora la coppia $\langle \delta(q_0, a), \delta'(q'_0, a) \rangle$ viene posta nella cella della seconda riga corrispondente alla prima colonna; altrimenti, si conclude che $\boldsymbol{A}$ e $\boldsymbol{A}'$ non sono equivalenti. Nel caso che l'esame abbia dato esito positivo, è la coppia $\langle \delta(q_0, b), \delta'(q'_0, b) \rangle$ ad essere esaminata a meno che non sia già stata esaminata. Se anche l'esame della coppia $\langle \delta(q_0, b), \delta'(q'_0, b) \rangle$ dà esito positivo, allora essa viene posta nella cella della terza riga corrispondente alla prima colonna. A questo punto, si completa la seconda riga con le stesse modalità viste per il completamento della prima riga e così via. L'algoritmo termina se l'esame di una coppia di stati dà esito negativo oppure non ci sono più righe della tabella da completare. Nel primo caso si conclude che $\boldsymbol{A}$ e $\boldsymbol{A}'$ non sono equivalenti, nell'altro che $\boldsymbol{A}$ e $\boldsymbol{A}'$ lo sono.

Esempio 6.2 Consideriamo i due automi finiti deterministici completi $\boldsymbol{A}$ e $\boldsymbol{A}'$ su $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ le cui tabelle di transizioni sono qui di seguito riportate.

q	a	b
→○ 0	0	1
1	2	0
2	1	2

Tabella delle transizioni di $\boldsymbol{A}$

q'	a	b
→○ $0'$	$0'$	$1'$
$1'$	$2'$	$3'$
$2'$	$0'$	$1'$
$3'$	$2'$	$3'$

Tabella delle transizioni di $\boldsymbol{A}'$

Quando applichiamo l'algoritmo di Moore otteniamo la tabella d Moore

$\langle q, q' \rangle$	$\langle \delta(q, \mathtt{a}), \delta'(q', \mathtt{a}) \rangle$	$\langle \delta(q, \mathtt{b}), \delta'(q', \mathtt{b}) \rangle$
$\langle 0, 0 \rangle$	$\langle 0, 0 \rangle$	$\langle 1, 1 \rangle$
$\langle 1, 1 \rangle$	$\langle 2, 2 \rangle$	$\langle 0, 3 \rangle$
$\langle 2, 2 \rangle$		

Nella fase di completamento della seconda riga, l'esame della coppia $\langle 0, 3 \rangle$ dà esito negativo cosicché si conclude che $\boldsymbol{A}$ e $\boldsymbol{A}'$ non sono equivalenti. ■

Esempio 6.3 Consideriamo i due automi finiti deterministici $\boldsymbol{A}$ e $\boldsymbol{A}'$ su $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ le cui tabelle delle transizioni sono qui di seguito riportate.

q	a	b
→ 0	1	2
1	1	3
2	1	2
3	1	4
○ 4	1	2

Tabella delle transizioni di $\boldsymbol{A}$

q'	a	b
→ 0′	1′	0′
1′	1′	2′
2′	1′	3′
○ 3′	1′	0′

Tabella delle transizioni di $\boldsymbol{A}'$

I due AFD sono equivalenti. Per provarlo, applichiamo l'algoritmo di Moore che termina dopo aver costruito la seguente tabella di Moore

$\langle q, q' \rangle$	$\langle \delta(q, \mathtt{a}), \delta'(q', \mathtt{a}) \rangle$	$\langle \delta(q, \mathtt{b}), \delta'(q', \mathtt{b}) \rangle$
$\langle 0, 0' \rangle$	$\langle 1, 1' \rangle$	$\langle 2, 0' \rangle$
$\langle 1, 1' \rangle$	$\langle 1, 1' \rangle$	$\langle 3, 2' \rangle$
$\langle 2, 0' \rangle$	$\langle 1, 1' \rangle$	$\langle 2, 0' \rangle$
$\langle 3, 2' \rangle$	$\langle 1, 1' \rangle$	$\langle 4, 3' \rangle$
$\langle 4, 3' \rangle$	$\langle 1, 1' \rangle$	$\langle 2, 0' \rangle$

con cinque righe complete. ■

6.1.2 *Minimizzazione di un automa finito deterministico*

Ora ci poniamo il problema di trovare, dato un AFD $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$, un AFD equivalente ad $\boldsymbol{A}$ e di dimensione minima.

A risolvere questo problema ci aiuta la seguente relazione di equivalenza tra stati di $\boldsymbol{A}$. Due stati q_1 e q_2 di $\boldsymbol{A}$ sono *indistinguibili* se sono equivalenti i due AFD $\boldsymbol{A}_1 = (Q, \Sigma, \delta, q_1, F)$ e $\boldsymbol{A}_2 = (Q, \Sigma, \delta, q_2, F)$; altrimenti, q_1 e q_2 sono *distinguibili*. Ad esempio, se $q_1 \in F$ e $q_2 \notin F$, allora q_1 e q_2 sono sicuramente distinguibili perché la stringa vuota è accettata da $\boldsymbol{A}_1$ ma non da $\boldsymbol{A}_2$.

Per ogni stato q di $\boldsymbol{A}$, indichiamo con $[q]$ la classe di equivalenza a cui appartiene q e consideriamo l'AFD $\boldsymbol{D} = (Q_{\boldsymbol{D}}, \Sigma, \delta_{\boldsymbol{D}}, [q_0], F_{\boldsymbol{D}})$, dove

> $Q_{\boldsymbol{D}}$ denota la partizione di Q indotta dalla relazione di indistinguibilità,
>
> $\delta_{\boldsymbol{D}}([q], a) := [\delta(q, a)]$ dove q è un qualsiasi stato appartenente a $[q]$,
>
> $F_{\boldsymbol{D}}$ è l'insieme delle classi di equivalenza a cui appartengono gli stati di accettazione di $\boldsymbol{A}$, cioè $F_{\boldsymbol{D}} = \{[q]: q \in F\}$.

Si può dimostrare che $\boldsymbol{D}$ è equivalente ad $\boldsymbol{A}$ e che, tra gli AFD equivalenti ad $\boldsymbol{A}$, $\boldsymbol{D}$ ha il numero di stati minimo. Dal punto di vista computazionale, la partizione $Q_{\boldsymbol{D}}$ di Q può essere ottenuta esaminando tutte le coppie degli stati di $\boldsymbol{A}$ e raggruppando assieme gli stati che risultano essere tra loro indistinguibili. Si osservi che, per sapere se due stati q_1 e q_2 di $\boldsymbol{A}$ sono indistinguibili, si potrebbe applicare l'algoritmo di Moore ai due AFD $\boldsymbol{A}_1 = (Q, \Sigma, \delta, q_1, F)$ e $\boldsymbol{A}_2 = (Q, \Sigma, \delta, q_2, F)$. Però, questa procedura per costruire la partizione $Q_{\boldsymbol{D}}$ è piuttosto laboriosa. Una più efficiente è la seguente. Se $F = \varnothing$ oppure $Q = F$, allora $Q_{\boldsymbol{D}}$ è la partizione banale di Q, cioè $Q_{\boldsymbol{D}} = \{Q\}$. Assumiamo che $F \neq \varnothing$ e $Q \neq F$. Inizialmente, $Q_{\boldsymbol{D}}$ è posta uguale alla bipartizione $\{F, Q\backslash F\}$. Poi, la partizione $Q_{\boldsymbol{D}}$ viene progressivamente raffinata alla maniera seguente: se un insieme X in $Q_{\boldsymbol{D}}$ con $|X| > 1$ si trova a contenere due stati q_1 e q_2 tali che, per qualche carattere a di Σ, gli stati $\delta(q_1, a)$ e $\delta(q_2, a)$ appartengono a due diverse classi di $Q_{\boldsymbol{D}}$, allora X viene scomposto in due sottoinsiemi disgiunti che contengono uno q_1 e l'altro q_2. Il processo di raffinamento va avanti finché la partizione $Q_{\boldsymbol{D}}$ non può essere raffinata ulteriormente ed allora $Q_{\boldsymbol{D}}$ viene a coincidere proprio con la partizione di Q indotta dalla relazione di indistinguibilità ed è tale che, per ogni classe X di $Q_{\boldsymbol{D}}$, si ha che o $X \subseteq F$ oppure $X \cap F = \varnothing$.

Esempio 6.4 Consideriamo di nuovo l'AFD dell'Esempio 6.1. La sua tabella delle transizioni è

q	a	b
→ 0	1	2
1	1	3
2	1	2
3	1	4
○ 4	1	2

Tabella delle transizioni di $\boldsymbol{A}$

Siccome $F = \{4\}$, la partizione $Q_{\boldsymbol{D}}$ è inizialmente formata dalle sole due classi $X = \{0, 1, 2, 3\}$ e $\{4\}$. Consideriamo la sottotabella della tabella delle transizioni indotta da X ma nella seconda e terza colonna riportiamo, anziché gli stati, le classi di $Q_{\boldsymbol{D}}$ a cui appartengono.

q	a	b
0	X	X
1	X	X
2	X	X
3	X	{4}

Sottotabella delle transizioni di $\boldsymbol{A}$ indotta da X

Pertanto, X viene scomposto nelle due classi: $Y = \{0, 1, 2\}$ e $\{3\}$. Consideriamo a questo punto la sottotabella della tabella delle transizioni indotta da Y.

q	a	b
0	Y	Y
1	Y	{3}
2	Y	Y

Sottotabella delle transizioni di $\boldsymbol{A}$ indotta da Y

Pertanto, Y viene scomposto nelle due classi: $Z = \{0, 2\}$ e $\{1\}$. Consideriamo infine la sottotabella della tabella delle transizioni indotta da Z.

q	a	b
0	{1}	Z
2	{1}	Z

Sottotabella delle transizioni di $\boldsymbol{A}$ indotta da Z

Pertanto, Z non viene scomposto e otteniamo così $Q_{\boldsymbol{D}} = \{\{0, 2\}, \{1\}, \{3\}, \{4\}\}$, e $\boldsymbol{D}$ è l'AFD specificato dalla tabella delle transizioni

$[q]$	a	b
→ {0, 2}	{1}	{0, 2}
{1}	{1}	{3}
{3}	{1}	{4}
○ {4}	{1}	{0, 2}

Tabella delle transizioni di $\boldsymbol{D}$

Il diagramma delle transizioni di $\boldsymbol{D}$ è mostrato in figura.

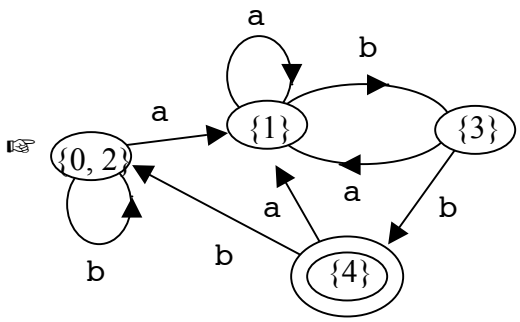


Diagramma delle transizioni di $\boldsymbol{D}$

Si osservi che (a meno di una ridenominazione degli stati) $\boldsymbol{D}$ è identico all'AFD $\boldsymbol{A}'$ dell'Esempio 6.3. ■

Una volta ottenuto un AFD $\boldsymbol{D}$ di dimensione minima, possiamo talora ridurre la dimensione di $\boldsymbol{D}$ eliminando i suoi *stati morti*, cioè gli stati $[q]$ di $\boldsymbol{D}$ distinti dallo stato iniziale $[q_0]$ di $\boldsymbol{D}$ tali che nel diagramma delle transizioni di $\boldsymbol{D}$

— nessuno stato di accettazione di $\boldsymbol{D}$ è raggiungibile da $[q]$, oppure
— $[q]$ non è raggiungibile dallo stato iniziale $[q_0]$ di $\boldsymbol{D}$.

Si osservi che quello che si ottiene non è detto che sia sempre un AFD, perché ora $|\delta([q], a)| \leq 1$ per ogni $([q], a) \in Q_{\boldsymbol{D}} \times \Sigma$. Lo chiameremo un automa finito *quasi-deterministico*.

6.2 Automi finiti nondeterministici

Un *automa finito nondeterministico* (AFN) è definito da una quintupla $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$ dove

Q è un insieme finito i cui elementi sono chiamati gli *stati* di $\boldsymbol{A}$;

Σ è un alfabeto;

δ (*funzione di transizione* di ***A***) è una funzione definita su $Q \times \Sigma$ con codominio l'insieme delle parti $\wp(Q)$ di Q;
q_0 è uno stato particolare, chiamato *iniziale* (o di *partenza*) di ***A***;

F è un sottoinsieme di Q, i cui elementi sono detti *stati finali* (o *di accettazione*) di ***A***.

Siccome ora i valori che può assumere la funzione di transizione δ sono sottoinsiemi (eventualmente vuoti) di Q, modifichiamo la definizione di transizione di ***A*** come segue. Una *transizione* di ***A*** è una terna (q, a, q') tale che $q' \in \delta(q, a)$. A questo punto, restano ben definite anche la tabella delle transizioni, il diagramma delle transizioni e la dimensione di ***A***. Si osservi che, a causa del nondeterminismo, la dimensione di ***A*** è $O(|Q|^2 \cdot |\Sigma|)$.

Veniamo ora alla *funzione di transizione estesa* $\hat{\delta}$ di ***A***: questa ha dominio $Q \times \Sigma^*$ e codominio $\wp(Q)$ ed è definita in maniera induttiva sulla lunghezza della stringa su Σ come segue:

per ogni stato q di ***A*** $\qquad \hat{\delta}(q, \varepsilon) = \{q\}$

per ogni stato q di ***A***, per ogni stringa y e per ogni carattere a

$$\hat{\delta}(q, ya) = \bigcup_{p \in \hat{\delta}(q, y)} \delta(p, a)$$

Si osservi che, come nel caso degli AFD, si ha sempre $\hat{\delta}(q, a) = \delta(q, a)$.

Una stringa x su Σ è *accettata* da ***A*** se e solo se $\hat{\delta}(q_0, x)$ contiene almeno uno stato di accettazione di ***A***; cioè, il *linguaggio accettato* da ***A*** è

$$L(\boldsymbol{A}) = \{x \in \Sigma^* : \hat{\delta}(q_0, x) \cap F \neq \varnothing\}.$$

Dunque, per decidere se una stringa x appartiene o meno a $L(\boldsymbol{A})$ abbiamo da calcolare l'insieme degli stati $\hat{\delta}(q_0, x)$. Sia $x = a_1 \ldots a_k$ con $k > 0$. Consideriamo il passo h.esimo, $1 \leq h \leq k$. Sia y il prefisso di x di lunghezza h–1; per ogni stato q in $\hat{\delta}(q_0, y)$, verrà calcolato l'insieme $\delta(q, a_h)$ cosicché il numero di transizioni effettuate dall'automa ***A*** è

$$\sum_{q \in \hat{\delta}(q_0, y)} |\delta(q, a_h)|$$

ed è, quindi, $O(|Q|^2)$. È conveniente far uso di due pile: una per l'insieme $\hat{\delta}(q_0, y)$ e l'altra per l'insieme $\hat{\delta}(q_0, ya_h)$, nonché un vettore binario indicizzato da Q per evitare di impilare due volte lo stesso stato.

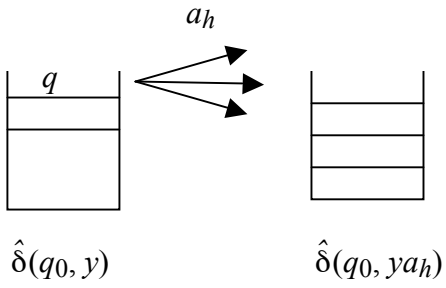


tabella delle transizioni

Dunque, il numero totale di transizioni di $\boldsymbol{A}$ per l'esame di x è $O(|x| \cdot |Q|^2)$, che ora dipende dal numero degli stati di $\boldsymbol{A}$.

Dimostreremo ora che il formalismo degli AFD ha lo stesso potere espressivo degli AFN. A tale scopo, diciamo che due AFN $\boldsymbol{A}$ e $\boldsymbol{A}'$ sono *equivalenti* se $L(\boldsymbol{A}) = L(\boldsymbol{A}')$.

Teorema 6.1 La classe dei linguaggi accettati da AFD coincide con classe dei linguaggi accettati da AFN.
Dimostrazione. Siccome ogni AFD è un AFN, basta provare che, per ogni AFN $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$, esiste un AFD ad esso equivalente. Se $\boldsymbol{A}$ è esso stesso un AFD, la tesi è ovvia. Dunque, escludiamo il caso che $\boldsymbol{A}$ sia un AFD.
Il caso più semplice è allora quello in cui l'automa $\boldsymbol{A}$ sia quasi-deterministico. In tal caso, un AFD equivalente ad $\boldsymbol{A}$ è $\boldsymbol{D} = (Q_{\boldsymbol{D}}, \Sigma, \delta_{\boldsymbol{D}}, q_0, F)$ con

$Q_{\boldsymbol{D}} = Q \cup \{\perp\}$ dove $\perp$ è un elemento che non appartiene a Q
$\delta_{\boldsymbol{D}}$ una funzione totale definita su $Q_{\boldsymbol{D}} \times \Sigma$ alla maniera seguente:

— per ogni $(q, a) \in Q \times \Sigma$,
se $\delta(q, a) = \varnothing$, allora $\delta_{\boldsymbol{D}}(q, a) := \perp$; altrimenti, $\delta_{\boldsymbol{D}}(q, a) := \delta(q, a)$;
— per ogni $a \in \Sigma$, $\delta_{\boldsymbol{D}}(\perp, a) := \perp$.

Consideriamo ora il caso generale e sia $\boldsymbol{D} = (Q_{\boldsymbol{D}}, \Sigma, \delta_{\boldsymbol{D}}, \{q_0\}, F_{\boldsymbol{D}})$ l'AFD così definito:

$Q_{\boldsymbol{D}} = \wp(Q)$
$\delta_{\boldsymbol{D}}(q_{\boldsymbol{D}}, a) = \bigcup_{q \in q_{\boldsymbol{D}}} \delta(q, a)$ per ogni stato $q_{\boldsymbol{D}}$ di $\boldsymbol{D}$
$F_{\boldsymbol{D}} = \{q_{\boldsymbol{D}} \in Q_{\boldsymbol{D}} : q_{\boldsymbol{D}} \cap F \neq \varnothing\}$.

Per dimostrare l'equivalenza di $\boldsymbol{A}$ e $\boldsymbol{D}$, dobbiamo provare che $\boldsymbol{A}$ accetta una stringa se e solo se la accetta $\boldsymbol{D}$. Sia x una qualsiasi stringa su Σ. Basterà provare che

$$\hat{\delta}(q_0, x) = \hat{\delta}_{\boldsymbol{D}}(\{q_0\}, x)$$

perché, allora, $\hat{\delta}(q_0, x) \cap F \neq \varnothing$ se e solo se $\hat{\delta}_{\boldsymbol{D}}(\{q_0\}, x) \in F_{\boldsymbol{D}}$. La dimostrazione è per induzione sulla lunghezza di x.

PASSO BASE. Se $x = \varepsilon$ allora $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$ e $\hat{\delta}_{\boldsymbol{D}}(\{q_0\}, \varepsilon) = \{q_0\}$ e la tesi è ovvia.

PASSO INDUTTIVO. Sia $x = ya$. Così,

$$\hat{\delta}(q_0, x) = \bigcup_{q \in \hat{\delta}(q_0, y)} \delta(q, a)$$

$$\hat{\delta}_{\boldsymbol{D}}(\{q_0\}, x) = \delta_{\boldsymbol{D}}(\hat{\delta}_{\boldsymbol{D}}(\{q_0\}, y), a) = \bigcup_{q \in \hat{\delta}_{\boldsymbol{D}}(\{q_0\}, y)} \delta(q, a)$$

D'altra parte, per l'ipotesi induttiva, si ha che

$$\hat{\delta}(q_0, y) = \hat{\delta}_{\boldsymbol{D}}(\{q_0\}, y)$$

da cui segue $\hat{\delta}(q_0, x) = \hat{\delta}_{\boldsymbol{D}}(\{q_0\}, x)$. □

In virtù del Lemma 6.1, il Teorema 6.1 ha il seguente corollario.

Corollario 6.1 Ogni linguaggio accettato da un automa finito nondeterministico è regolare.

Dato un AFN $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$, per costruire un AFD equivalente ad $\boldsymbol{A}$ possiamo far ricorso al metodo utilizzato nella dimostrazione del Teorema 6.1 ma il numero degli stati dell'AFD così costruito è pari a $2^{|Q|}$. L'algoritmo che ora daremo costruisce un AFD $\boldsymbol{D} = (Q_{\boldsymbol{D}}, \Sigma, \delta_{\boldsymbol{D}}, \{q_0\}, F_{\boldsymbol{D}})$ che è equivalente ad $\boldsymbol{A}$ ed ha un numero "ragionevole" di stati.

Algoritmo 6.1

1. $Q_{\boldsymbol{D}} := \{\{q_0\}\}$; dichiarare lo stato $\{q_0\}$ in $Q_{\boldsymbol{D}}$ "aperto".

2. Fintantoché esiste uno stato aperto $q_{\boldsymbol{D}}$ in $Q_{\boldsymbol{D}}$, ripetere

    dichiarare $q_{\boldsymbol{D}}$ "chiuso";
    per ogni $a \in \Sigma$, ripetere
        $p := \bigcup_{q \in q_{\boldsymbol{D}}} \delta(q, a)$;
        se $p \notin Q_{\boldsymbol{D}}$, allora
            aggiungere p a $Q_{\boldsymbol{D}}$;
            dichiarare p "aperto";
        $\delta_{\boldsymbol{D}}(q_{\boldsymbol{D}}, a) := p$.

3. $F_{\boldsymbol{D}} := \{ q_{\boldsymbol{D}} \in Q_{\boldsymbol{D}} : q_{\boldsymbol{D}} \cap F \neq \varnothing \}$.

Si dimostra facilmente che l'automa finito deterministico $\boldsymbol{D}$ così costruito è equivalente all'AFN $\boldsymbol{A}$. Inoltre, "spesso" $|Q_{\boldsymbol{D}}|$ è polinomiale in $|Q|$, anche se nel caso generale $|Q_{\boldsymbol{D}}|$ è dell'ordine di $2^{|Q|}$.

Esempio 6.5 Consideriamo l'AFN $\boldsymbol{A}$ su $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ con tabella delle transizioni

q	a	b
→ 0	{0, 1}	{0}
1	∅	{2}
2	∅	{3}
○ 3	∅	∅

Tabella delle transizioni di $\boldsymbol{A}$

Il diagramma delle transizioni di $\boldsymbol{A}$ è mostrato in figura.

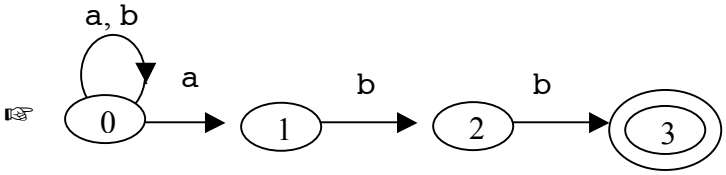


Diagramma delle transizioni di $\boldsymbol{A}$

Per ottenere un AFD ***D*** equivalente ad ***A***, utilizziamo l'Algoritmo 6.1 e otteniamo l'AFD ***D*** con la seguente tabella delle transizioni

q_D	a	b
→ {0}	{0, 1}	{0}
{0, 1}	{0, 1}	{0, 2}
{0, 2}	{0, 1}	{0, 3}
○ {0, 3}	{0, 1}	{0}

Tabella delle transizioni di ***D***

Il diagramma delle transizioni di ***D*** è mostrato in figura.

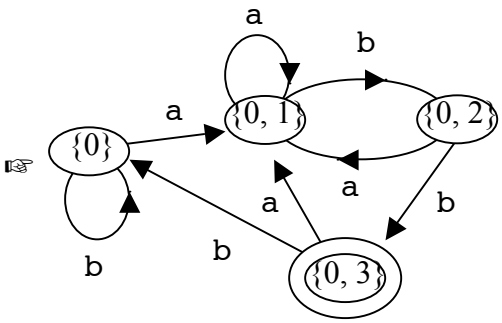


Diagramma delle transizioni di ***D*** ■

Esempio 6.6 Consideriamo l'AFN ***A*** su Σ = {a, b} con tabella delle transizioni

q	a	b
→ 0	{0, 1}	{0}
1	{2}	{2}
2	{3}	{3}
3	{4}	{4}
○ 4	∅	∅

Tabella delle transizioni di ***A***

Se applichiamo l'Algoritmo 6.1, otteniamo un AFD ***D*** con ben sedici ($= 2^{|Q|-1}$) stati, la cui tabella delle transizioni è qui riportata.

q_D	a	b
→ {0}	{0, 1}	{0}
{0, 1}	{0, 1, 2}	{0, 2}
{0, 2}	{0, 1, 3}	{0, 3}
{0, 3}	{0, 1, 4}	{0, 4}

○	{0, 4}	{0, 1}	{0}
	{0, 1, 2}	{0, 1, 2, 3}	{0, 2, 3}
	{0, 1, 3}	{0, 1, 2, 4}	{0, 2, 4}
○	{0, 1, 4}	{0, 1, 2}	{0, 2}
	{0, 2, 3}	{0, 1, 3, 4}	{0, 3, 4}
○	{0, 2, 4}	{0, 1, 3}	{0, 3}
○	{0, 3, 4}	{0, 1, 4}	{0, 4}
	{0, 1, 2, 3}	{0, 1, 2, 3, 4}	{0, 2, 3, 4}
○	{0, 1, 2, 4}	{0, 1, 2, 3}	{0, 2, 3}
○	{0, 1, 3, 4}	{0, 1, 2, 4}	{0, 2, 4}
○	{0, 2, 3, 4}	{0, 1, 3, 4}	{0, 3, 4}
○	{0, 1, 2, 3, 4}	{0, 1, 2, 3, 4}	{0, 2, 3, 4}

Tabella delle transizioni di ***D*** ■

Si osservi infine che il Teorema 6.1 fornisce un'alternativa alla procedura di riconoscimento delle stringhe accettate da un AFN. Prima si costruisce un AFD equivalente all'AFN (con l'Algoritmo 6.1), quindi lo si minimizza e infine si applica la procedura di riconoscimento per l'AFD ottenuto. Così facendo, accettare o rifiutare una stringa richiede un tempo lineare nella sua lunghezza; il rischio che si corre però è che il numero degli stati dell'AFD sia esponenziale nel numero degli stati dell'AFN.

6.3 Automi finiti nondeterministici con transizioni spontanee

Una *transizione spontanea* è una terna del tipo (q, ε, q') con $q \neq q'$. Un *AFN con transizioni spontanee* (o un ε-*AFN*) è definito da una quintupla $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$ dove

Q è un insieme finito i cui elementi sono chiamati gli *stati* di $\boldsymbol{A}$;

Σ è un alfabeto;

δ (*funzione di transizione* di $\boldsymbol{A}$) è una funzione definita su $Q \times (\Sigma \cup \{\varepsilon\})$ con codominio $\wp(Q)$;

q_0 è uno stato particolare, chiamato *stato iniziale* (o di *partenza*) di $\boldsymbol{A}$;

F è un sottoinsieme di Q, i cui elementi sono detti *stati finali* (o *di accettazione*) di $\boldsymbol{A}$.

Di nuovo, una *transizione* di $\boldsymbol{A}$ è una terna (q, a, q') tale che $q' \in \delta(q, a)$. Le definizioni della tabella delle transizioni, del diagramma delle transizioni e della dimensione di un ε-AFN sono del tutto analoghe a quelle di un AFN.

Esempio 6.7 Consideriamo l'ε-AFN $\boldsymbol{A}$ su $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ con $Q = \{0, 1, 2, \ldots, 23\}$ e con la seguente tabella delle transizioni

q	a	b	ε
→ 0	∅	∅	{1, 7}
1	∅	∅	{2, 3}
2	{4}	∅	∅
3	∅	{5}	∅
4	∅	∅	{6}
5	∅	∅	{6}
6	∅	∅	{1, 7}
7	{8}	∅	∅
8	∅	∅	{9, 10}
9	{11}	∅	∅
10	∅	{12}	∅
11	∅	∅	{13}
12	∅	∅	{13}
13	∅	∅	{14, 15}
14	{16}	∅	∅
15	∅	{17}	∅
16	∅	∅	{18}
17	∅	∅	{18}
18	∅	∅	{19, 20}
19	{21}	∅	∅
20	∅	{22}	∅
21	∅	∅	{23}
22	∅	∅	{23}
○ 23	∅	∅	∅

Tabella delle transizioni di $\boldsymbol{A}$ ■

Uno stato q' di $\boldsymbol{A}$ è *raggiungibile spontaneamente* da uno stato q di $\boldsymbol{A}$ se $q' = q$ oppure esistono n transizioni spontanee (q_1, ε, q_2), …, $(q_n, \varepsilon, q_{n+1})$, $n \geq 1$, tali che $q_1 = q$ e $q_{n+1} = q'$. La ε-*chiusura* di uno stato q, che indichiamo con $[q]_\varepsilon$, è l'insieme formato dagli stati che sono raggiungibili spontaneamente da q. Più in generale, la ε-chiusura di un insieme di stati X è definito dall'insieme

$$[X]_\varepsilon = \cup_{q \in X} [q]_\varepsilon$$

con la convenzione $[\varnothing]_\varepsilon = \varnothing$. Si noti che, nel grafo orientato che si ottiene dal diagramma delle transizioni ignorando tutti gli archi la cui etichetta non contiene la stringa vuota, l'insieme $[X]_\varepsilon$ è formato da quei vertici che sono raggiungibili da almeno un vertice appartenente ad X.

Esempio 6.7 (seguito) La seguente tabella riporta la ε-chiusura di ogni stato q dell'ε-AFN $\boldsymbol{A}$.

q	$[q]_\varepsilon$
0	{0, 1, 2, 3, 7}
1	{1, 2, 3}
2	{2}
3	{3}
4	{1, 2, 3, 4, 6, 7}
5	{1, 2, 3, 5, 6, 7}
6	{1, 2, 3, 6, 7}
7	{7}
8	{8, 9, 10}
9	{9}
10	{10}
11	{11, 13, 14, 15}
12	{12, 13, 14, 15}
13	{13, 14, 15}
14	{14}
15	{15}
16	{16, 18, 19, 20}
17	{17, 18, 19, 20}
18	{18, 19, 20}
19	{19}
20	{20}
21	{21, 23}
22	{22, 23}
23	{23}

Le ε-chiusure degli stati di $\boldsymbol{A}$ ■

La *funzione di transizione estesa* $\hat{\delta}$ di $\boldsymbol{A}$ è la funzione con dominio $Q \times \Sigma^*$ e codominio $\wp(Q)$ così definita in maniera induttiva sulla lunghezza della stringa su Σ:

per ogni stato q di $\boldsymbol{A}$ $\quad\quad \hat{\delta}(q, \varepsilon) = [q]_\varepsilon$

per ogni stato q di $\boldsymbol{A}$, per ogni stringa y e per ogni carattere a

$$\hat{\delta}(q, ya) = \bigcup_{p \in \hat{\delta}(q, y)} [\delta(p, a)]_\varepsilon \ .$$

Si osservi che, a differenza degli AFN, si ha sempre

$$\hat{\delta}(q, a) = \bigcup_{p \in [q]_\varepsilon} [\delta(p, a)]_\varepsilon \ ;$$

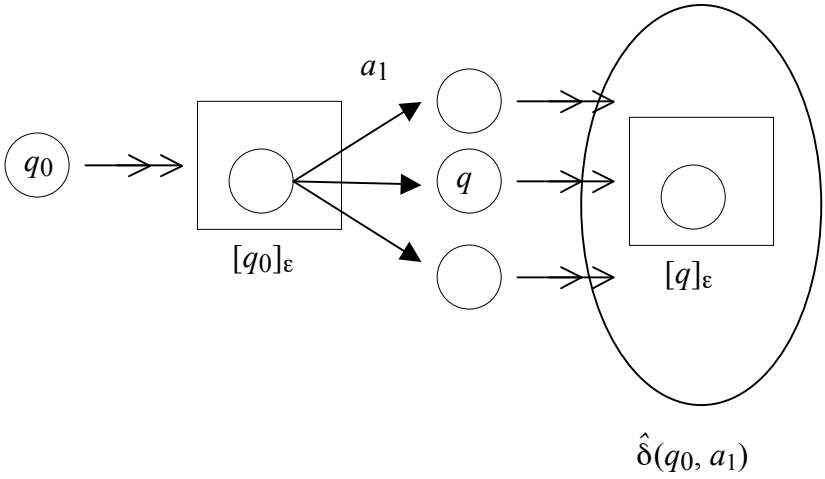
inoltre

$$\hat{\delta}(q, \varepsilon) = [\delta(q, \varepsilon)]_\varepsilon \ .$$

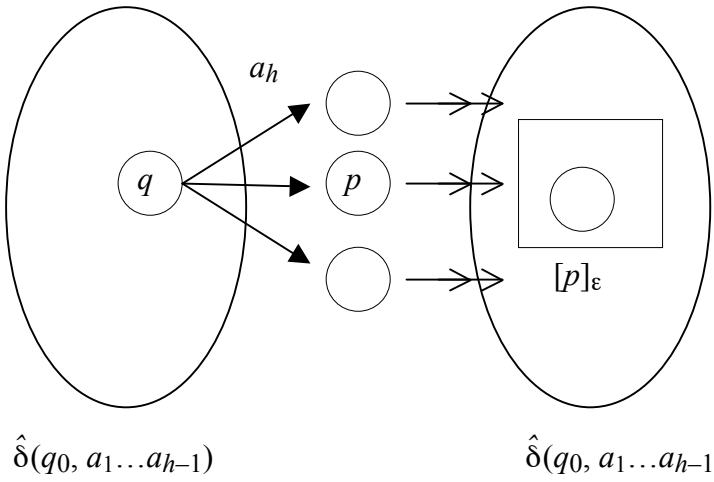
Una stringa x su Σ è *accettata* da $\boldsymbol{A}$ se e solo se $\hat{\delta}(q_0, x)$ contiene almeno uno stato di accettazione di $\boldsymbol{A}$; cioè, il linguaggio accettato da $\boldsymbol{A}$ è

$$L(\boldsymbol{A}) = \{x \in \Sigma^* : \hat{\delta}(q_0, x) \cap F \neq \varnothing\}.$$

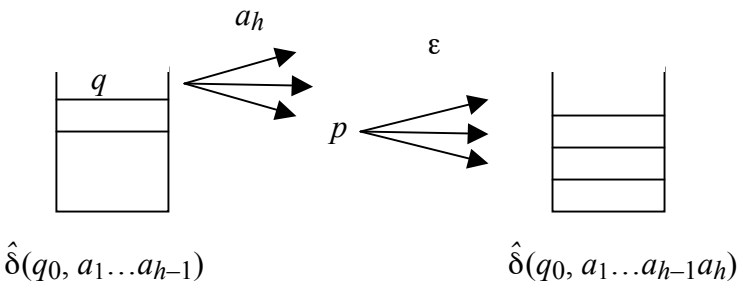
Dunque, per decidere se una stringa x appartiene o meno a $L(\boldsymbol{A})$ abbiamo da calcolare l'insieme degli stati $\hat{\delta}(q_0, x)$. Sia $x = a_1 \ldots a_k$ con $k > 0$. Consideriamo il primo passo. La figura illustra come viene calcolato l'insieme $\hat{\delta}(q_0, a_1)$.



Consideriamo ora il generico passo h.esimo, $1 < h \leq k$. Sia y il prefisso di x di lunghezza h–1; per ogni stato q in $\hat{\delta}(q_0, a_1 \ldots a_{h-1})$, verrà prima calcolato l'insieme $\delta(q, a_h)$ dopodiché verrà calcolata la sua ε-chiusura).



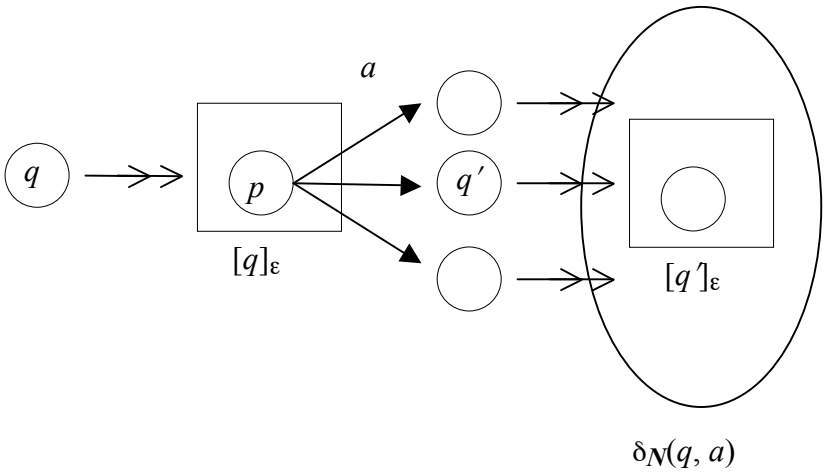
Anche qui è conveniente far uso di due pile: una per l'insieme $\hat{\delta}(q_0, a_1 \ldots a_{h-1})$ e l'altra per l'insieme $\hat{\delta}(q_0, a_1 \ldots a_{h-1}a_h)$, nonché un vettore binario indicizzato da Q per evitare di impilare due volte lo stesso stato.



Dimostreremo ora che il formalismo degli AFN ha lo stesso potere espressivo degli ε-AFN. A tale scopo diciamo che due ε-AFN $\boldsymbol{A}$ e $\boldsymbol{A}'$ sono *equivalenti* se $L(\boldsymbol{A}) = L(\boldsymbol{A}')$.

Teorema 6.2 La classe dei linguaggi accettati dagli AFN coincide con classe dei linguaggi accettati dagli ε-AFN.
Dimostrazione. Siccome ogni AFN è un ε-AFN, basta provare che, per ogni ε-AFN $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$, esiste un AFN equivalente ad $\boldsymbol{A}$. Se $\boldsymbol{A}$ è esso stesso un AFN, la tesi è ovvia. Dunque, escludiamo il caso che $\boldsymbol{A}$ sia un AFN. Consideriamo allora l'AFN $\boldsymbol{N} = (Q, \Sigma, \delta_{\boldsymbol{N}}, q_0, F_{\boldsymbol{N}})$ con

— $\delta_{\boldsymbol{N}}(q, a) = \bigcup_{p \in [q]_\varepsilon} [\delta(p, a)]_\varepsilon$



— $F_{\boldsymbol{N}} := \{q: [q]_\varepsilon \cap F \neq \varnothing\}$

Allora è ovvio che una stringa è accettata da $\boldsymbol{N}$ se e solo se è accettata da $\boldsymbol{A}$ così che $\boldsymbol{N}$ ed $\boldsymbol{A}$ sono equivalenti. □

Per il Corollario 6.1, il Teorema 6.2 ha il seguente corollario.

Corollario 6.2 Ogni linguaggio accettato da un automa finito nondeterministico con transizioni spontanee è regolare.

Esempio 6.7 (seguito) Per costruire un AFN equivalente all'ε-AFN $\boldsymbol{A}$, possiamo impiegare la tecnica usata nella dimostrazione del Teorema 6.2, e otteniamo l'AFD con la seguente tabella delle transizioni.

q	a	b
→ 0	{1, 2, 3, 4, 6, 7, 8, 9, 10}	{1, 2, 3, 5, 6, 7}
1	{1, 2, 3, 4, 6, 7}	{1, 2, 3, 5, 6, 7}
2	{1, 2, 3, 4, 6, 7}	∅
3	∅	{1, 2, 3, 5, 6, 7}
4	{1, 2, 3, 4, 6, 7, 8, 9, 10}	{1, 2, 3, 5, 6, 7}
5	{1, 2, 3, 4, 6, 7, 8, 9, 10}	{1, 2, 3, 5, 6, 7}
6	{1, 2, 3, 4, 6, 7, 8, 9, 10}	{1, 2, 3, 5, 6, 7}
7	{8, 9, 10}	∅
8	{11, 13, 14, 15}	{12, 13, 14, 15}
9	{11, 13, 14, 15}	∅
10	∅	{12, 13, 14, 15}
11	{16, 18, 19, 20}	{17, 18, 19, 20}
12	{16, 18, 19, 20}	{17, 18, 19, 20}
13	{16, 18, 19, 20}	{17, 18, 19, 20}
14	{16, 18, 19, 20}	∅
15	∅	{17, 18, 19, 20}
16	{21, 23}	{22, 23}
17	{21, 23}	{22, 23}
18	{21, 23}	{22, 23}
19	{21, 23}	∅
20	∅	{22, 23}
○ 21	∅	∅
○ 22	∅	∅
○ 23	∅	∅

Tabella delle transizioni di un AFD equivalente ad $\boldsymbol{A}$. ■

Dato un ε-AFN $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$, per costruire un AFD equivalente ad $\boldsymbol{A}$ possiamo costruire prima un AFN $\boldsymbol{N}$ equivalente ad $\boldsymbol{A}$ (v. dimostrazione del Teorema 6.2), e poi un AFD equivalente ad $\boldsymbol{N}$ (e, quindi, ad $\boldsymbol{A}$) con l'Algorimo 6.1. Una scorciatoia è offerta dalla seguente variante dell'Algoritmo 6.1, che costruisce direttamente da $\boldsymbol{A}$ un AFD $\boldsymbol{D} = (Q_{\boldsymbol{D}}, \Sigma, \delta_{\boldsymbol{D}}, (q_0)_\varepsilon, F_{\boldsymbol{D}})$ equivalente ad $\boldsymbol{A}$.

Algoritmo 6.2

1. $Q_{\boldsymbol{D}} := \{[q_0]_\varepsilon\}$; dichiarare lo stato $[q_0]_\varepsilon$ "aperto".

2. Fintantoché esiste uno stato aperto $q_{\boldsymbol{D}}$ in $Q_{\boldsymbol{D}}$, ripetere

    dichiarare $q_{\boldsymbol{D}}$ "chiuso";
    per ogni $a \in \Sigma$, ripetere
        $p := \bigcup_{q \in q_{\boldsymbol{D}}} [\delta(q, a)]_\varepsilon$;
        se $p \notin Q_{\boldsymbol{D}}$, allora
            aggiungere p a $Q_{\boldsymbol{D}}$;
            dichiarare p "aperto";
        $\delta_{\boldsymbol{D}}(q_{\boldsymbol{D}}, a) := p$.

3. $F_{\boldsymbol{D}} := \{ q_{\boldsymbol{D}} \in Q_{\boldsymbol{D}} : q_{\boldsymbol{D}} \cap F \neq \varnothing \}$.

Non è difficile dimostrare che l'AFD ***D*** così costruito è equivalente all'ε-AFN ***A***.

Esempio 6.8. Consideriamo l'ε-AFN ***A*** con tabella delle transizioni

q	a	b	ε
→ 0	∅	∅	{1, 7}
1	∅	∅	{2, 4}
2	{3}	∅	∅
3	∅	∅	{6}
4	∅	{5}	∅
5	∅	∅	{6}
6	∅	∅	{1, 7}
7	{8}	∅	∅
8	∅	{9}	∅
9	∅	{10}	∅
○ 10	∅	∅	∅

Tabella delle transizioni di ***A***

L'AFD ***D*** generato dall'Algoritmo 6.2 ha cinque stati

$q_{\boldsymbol{D}}$	sottoinsieme di Q
0*	{0, 1, 2, 4, 7}
1*	{1, 2, 3, 4, 6, 7, 8}
2*	{1, 2, 4, 5, 6, 7}
3*	{1, 2, 4, 5, 6, 7, 9}
4*	{1, 2, 4, 5, 6, 7, 10}

e la sua tabella delle transizioni è la seguente

q*	a	b
→ 0*	1*	2*
1*	1*	3*
2*	1*	2*
3*	1*	4*
○ 4*	1*	2*

Tabella delle transizioni di $\boldsymbol{D}$ ■

Ovviamente, come per l'Algoritmo 6.1, nel caso peggiore l'AFD costruito con l'Algoritmo 6.2 ha un numero di stati dell'ordine di $2^{|Q|}$. Per fortuna in molti casi non lo è e, allora , l'uso dell'AFD consente di risolvere il problema del riconoscimento delle stringhe accettate da un ε-AFN in tempo lineare.

A questo punto, non resta che dimostrare che, per ogni linguaggio regolare, esiste un automa finito che lo accetta. A tale scopo, introduciamo una "forma standard" per grammatiche regolari.
Una grammatica regolare $\boldsymbol{G} = (N, T, P, S)$ è in *forma standard* se ogni produzione nonnulla è della forma $A \rightarrow aB$, dove A e B sono simboli nonterminali e a è un simbolo terminale. In altri termini, sono escluse

— le produzioni unitarie $A \rightarrow B$,

— le produzioni multiple $A \rightarrow xB$ con $|x| > 1$,

— le produzioni $A \rightarrow x$ con $x \neq \varepsilon$.

Per eliminare le produzioni unitarie, possiamo utilizzare l'Algoritmo 2.4 visto nel processo di normalizzazione delle grammatiche acontestuali (v. paragrafo 2.3). Per eliminare le produzioni multiple procediamo alla maniera seguente.
Per ogni produzione multipla $A \rightarrow a_1 \ldots a_k B$:

introdurre k–1 nuovi simboli nonterminali $C_1, \ldots, C_{k-1}$;
eliminare la produzione $A \rightarrow a_1 \ldots a_k B$;
aggiungere le k–1 produzioni:

$A \rightarrow a_1 C_1$,
$C_1 \rightarrow a_2 C_2$,
…
$C_{k-1} \rightarrow a_k B$.

Per eliminare infine le produzioni del tipo $A \to a_1 \ldots a_k$ con $k \geq 1$, procediamo alla maniera seguente:

introdurre k nuovi simboli nonterminali $C_1, \ldots, C_k$;
eliminare la produzione $A \to a_1 \ldots a_k$;
aggiungere le k+1 produzioni:

$A \to a_1 C_1$,
$C_1 \to a_2 C_2$,
…,
$C_{k-1} \to a_k C_k$
$C_k \to \varepsilon$.

È immediato verificare che la grammatica che se ne ottiene è in forma standard ed è equivalente a quella originaria. Abbiamo così il seguente risultato.

Lemma 6.2 Per ogni grammatica regolare ne esiste una equivalente in forma standard.

Teorema 6.3 La classe dei linguaggi accettati da automi finiti nondeterministici con transizioni spontanee coincide con la classe dei linguaggi regolari.
Dimostrazione. Per il Corollario 6.2, basta provare che per ogni grammatica regolare $\boldsymbol{G} = (N, T, P, S)$ esiste un ε-AFN che accetta il linguaggio generato da $\boldsymbol{G}$. Grazie al Lemma 6.2, possiamo assumere che $\boldsymbol{G}$ sia in forma standard. Si consideri allora l'ε-AFN $\boldsymbol{A} = (Q, \Sigma, \delta, S, F)$ così definito:

$Q = N \cup \{f\}$, dove f è un simbolo che non appartiene a $N \cup T$;
$\Sigma = T$;
la funzione di transizione δ è così definita

— per ogni $(A, a) \in N \times T$, $\delta(A, a)$ è l'insieme dei simboli nonterminali B di $\boldsymbol{G}$ tali che $A \to aB$ sia una produzione di $\boldsymbol{G}$,
— per ogni $A \in N$, se $A \to \varepsilon$ è una produzione di $\boldsymbol{G}$ allora $\delta(A, \varepsilon) = \{f\}$, altrimenti $\delta(A, \varepsilon) = \varnothing$.
— per ogni $e \in T \cup \{\varepsilon\}$, $\delta(f, e) = \varnothing$;

$F = \{f\}$.

È allora ovvio che una stringa x è una frase di $\boldsymbol{G}$ se e solo se $\hat{\delta}(S, x) = \{f\}$, cioè se e solo se $\boldsymbol{A}$ accetta x. Abbiamo così provato che $L(\boldsymbol{A}) = L(\boldsymbol{G})$. □

Corollario 6.3 Sia la classe dei linguaggi accettati da automi finiti nondeterministici che la classe dei linguaggi accettati da automi finiti deterministici coincidono con la classe dei linguaggi regolari.

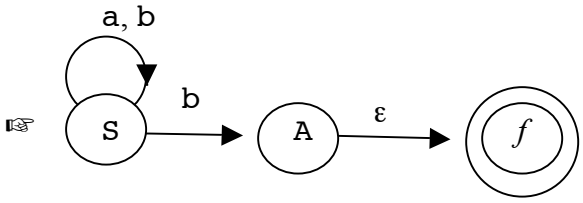
Esempio 6.9 Consideriamo la grammatica $\boldsymbol{G} = (N, T, P, \mathtt{S})$ con $N = \{\mathtt{S}\}$, $T = \{\mathtt{a}, \mathtt{b}\}$ e le tre produzioni

$$\mathtt{S} \rightarrow \mathtt{aS} \mid \mathtt{bS} \mid \mathtt{b}$$

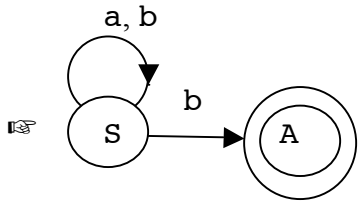
Una grammatica in forma standard equivalente a $\boldsymbol{G}$ è la grammatica $\boldsymbol{G}' = (N', T, P', \mathtt{S})$ con $N' = \{\mathtt{S}, \mathtt{A}\}$ e le quattro produzioni

$$\mathtt{S} \rightarrow \mathtt{aS} \mid \mathtt{bS} \mid \mathtt{bA}$$
$$\mathtt{A} \rightarrow \varepsilon$$

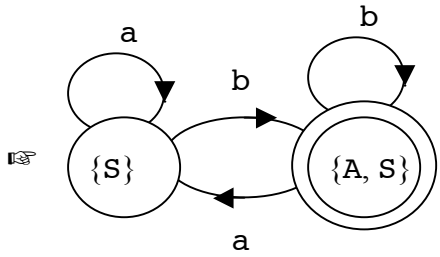
Un ε-AFN che accetta il linguaggio generato da $\boldsymbol{G}$ ha il seguente diagramma delle transizioni:



Un AFN che accetta il linguaggio generato da $\boldsymbol{G}$ ha il seguente diagramma delle transizioni:



Un AFD che accetta il linguaggio generato da $\boldsymbol{G}$ ha il seguente diagramma delle transizioni:



■

Cap. 7

ESPRESSIONI REGOLARI

Si è visto che per specificare un linguaggio lessicale (un linguaggio regolare) possiamo usare le grammatiche regolari oppure gli automi finiti. Tuttavia, né gli automi finiti né le grammatiche regolari forniscono una maniera compatta per denotare un linguaggio regolare. Per ottenerla, introdurremo un'algebra le cui "espressioni" forniscono una codifica compatta dei linguaggi regolari.

7.1 Linguaggi di Kleene

Si ricordi che i linguaggi su di un alfabeto Σ sono sottoinsiemi del linguaggio universale Σ^* cosicché l'insieme $\mathcal{L}(\Sigma)$ dei linguaggi su Σ coincide con l'insieme delle parti di Σ^*. Su $\mathcal{L}(\Sigma)$ possiamo definire le usuali operazioni insiemistiche di unione, intersezione e di complementazione che danno ad $\mathcal{L}(\Sigma)$ la struttura di un'algebra booleana. Come proveremo più in là, per i linguaggi regolari possiamo limitarci a considerare le sole tre operazioni: l'unione (+), la moltiplicazione ($\cdot$) e l'operatore di Kleene (*), queste ultime così definite:

— La *moltiplicazione* (o *concatenazione*) di due linguaggi L per M in $\mathcal{L}(\Sigma)$ dà come risultato il linguaggio su Σ così definito

$$L \cdot M = \{xy : x \in L \text{ e } y \in M\},$$

che chiamiamo il *prodotto* di L per M. Utilizzando la moltiplicazione, possiamo definire la *potenza n-esima* di L come:

$$L^n = \begin{cases} \{\varepsilon\} & \text{se } n = 0 \\ L \cdot L^{n-1} & \text{se } n > 0 \end{cases}$$

— L'operatore di Kleene applicato ad un linguaggio L in $\mathcal{L}(\Sigma)$ dà come risultato il linguaggio L^* così definito

$$L^* = \bigcup_{n = 0, \ldots, \infty} L^n .$$

Per le tre operazioni, d'ora innanzi adottiamo il seguente ordine di precedenza: (*, $\cdot$, +). Indichiamo con $\mathcal{K}(\Sigma)$ la più piccola classe dei linguaggi in $\mathcal{L}(\Sigma)$ che

- contiene il linguaggio vuoto $\varnothing$ e il linguaggio elementare $\{a\}$ per ogni carattere $a \in \Sigma$, e
- è chiusa rispetto alle tre operazioni +, $\cdot$ e *.

Chiamiamo *linguaggi di Kleene* gli elementi di $\mathcal{K}(\Sigma)$ e *algebra di Kleene* la struttura algebrica $(\mathcal{K}(\Sigma), +, \cdot, *)$. Si osservi che, siccome $\varnothing^* = \{\varepsilon\}$, $\mathcal{K}(\Sigma)$ contiene il linguaggio $\{\varepsilon\}$. Inoltre, siccome $\mathcal{K}(\Sigma)$ è chiusa rispetto alle operazioni + e *, $\mathcal{K}(\Sigma)$ contiene anche il linguaggio universale Σ^*. Infine, siccome $\mathcal{K}(\Sigma)$ è chiusa rispetto alle operazioni + e ·, se L è un linguaggio di Kleene, allora $\mathcal{K}(\Sigma)$ contiene anche il linguaggio

$$L^+ = \sum_{k = 1, \ldots, \infty} L^k .$$

Le operazioni + e · nell'algebra di Kleene godono delle proprietà seguenti:

$(L + M) + N = L + (M + N)$	associatività
$L + M = M + L$	commutatività
$L + \varnothing = \varnothing + L = L$	elemento neutro dell'unione
$(L \cdot M) \cdot N = L \cdot (M \cdot N)$	associatività
$L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$	elemento neutro della moltiplicazione
$L \cdot (M + N) = L \cdot M + L \cdot N$	distributività a destra
$(M + N) \cdot L = M \cdot L + N \cdot L$	distributività a sinistra

Inoltre valgono le seguenti identità algebriche:

$$L \cdot \varnothing = \varnothing \cdot L = \varnothing$$

$$L^* = \{\varepsilon\} + L^+$$

$$L^+ = L \cdot L^* = L^* \cdot L$$

$$L^* = (L^*)^* = L^* \cdot L^* = (\{\varepsilon\} + L)^* = \\ = L^* \cdot (\{\varepsilon\} + L) = (\{\varepsilon\} + L) \cdot L^* = \{\varepsilon\} + L \cdot L^*.$$

$$(L + M)^* = (L^* + M^*)^* = (L^* \cdot M^*)^* = (L^* \cdot M)^* \cdot L^* = L^* \cdot (M \cdot L^*)^*$$

$$L \cdot (M \cdot L)^* = (L \cdot M)^* \cdot L$$

$$(L^* \cdot M)^* = \{\varepsilon\} + (L + M)^* \cdot M$$

$$(L \cdot M^*)^* = \{\varepsilon\} + L \cdot (L + M)^*$$

$$L \cdot M^* = L + L \cdot (M + \{\varepsilon\})^* \cdot (M + \{\varepsilon\})$$

Molte di queste identità si dimostrano con la tecnica della "riaggregazione". A titolo illustrativo, consideriamo l'identità

$$L \cdot (M \cdot L)^* = (L \cdot M)^* \cdot L$$

e sia x una generica stringa appartente al linguaggio $L \cdot (M \cdot L)^*$. Allora, x è della forma

$$x = y_0(z_1y_1)(z_2y_2)\ldots(z_ny_n)$$

dove $y_0, y_1, y_2, \ldots, y_n$ appartengono ad L e $z_1, z_2, \ldots, z_n$ appartengono ad M. Visto che la moltiplicazione è associativa, possiamo riscrivere x nella forma

$$x = (y_0z_1)(y_1z_2)\ldots(y_{n-1}z_n)\, y_n$$

la qual cosa prova che x appartiene al linguaggio $(L \cdot M)^* \cdot L$ e, quindi, che il linguaggio $L \cdot (M \cdot L)^*$ è contenuto in $(L \cdot M)^* \cdot L$. In maniera analoga possiamo dimostrare che il linguaggio $(L \cdot M)^* \cdot L$ è contenuto in $L \cdot (M \cdot L)^*$, di qui l'identità.

7.2 Espressioni nell'algebra di Kleene

Introduciamo ora un formalismo per specificare i linguaggi di Kleene su un alfabeto Σ assegnato. Un'*espressione* nell'algebra di Kleene su Σ è una stringa sull'alfabeto

$$\Omega = \Sigma \cup \{\varnothing, |, *, (,)\}$$

così definita:

1. $\varnothing$ è un'espressione;
2. per ogni carattere a di Σ, a è un'espressione;
3. se E_1 e E_2 sono espressioni, lo sono anche (E_1), $(E_1)|(E_2)$, $(E_1)(E_2)$ ed $(E_1)^*$.

Il linguaggio *denotato* da un'espressione E nell'algebra di Kleene, indicato con $L(E)$, è il linguaggio di Kleene così definito:

— se $E = \varnothing$, allora $L(E) = \varnothing$,

— se $E = a$, allora $L(E) = \{a\}$,

— se $E = (E_1)$, allora $L(E) = L(E_1)$,

— se $E = (E_1)|(E_2)$, allora $L(E) = L(E_1) + L(E_2)$,

— se $E = (E_1)(E_2)$, allora $L(E) = L(E_1) \cdot L(E_2)$,

— se $E = (E_1)^*$, allora $L(E) = (L(E_1))^*$.

Va da sé che ogni linguaggio di Kleene è denotato da un'espressione nell'algebra di Kleene e che ogni espressione nell'algebra di Kleene denota un linguaggio di Kleene. Adottiamo poi le seguenti abbreviazioni:

$(E)^+$ per $(E)((E)^*)$
ε per $(\varnothing)^*$.

Due espressioni E_1 e E_2 nell'algebra di Kleene sono *equivalenti*, $E_1 \approx E_2$, se $L(E_1) = L(E_2)$. Così, a partire dalle proprietà e dalle identità menzionate, abbiamo ad esempio le seguenti equivalenze tra espressioni:

$(E) \approx E$
$((E_1)|(E_2))|(E_3) \approx (E_1)|((E_2)|(E_3))$
$(E_1)|(E_2) \approx (E_2)|(E_1)$
$(E)|(\varnothing) \approx (\varnothing)|(E) \approx E$

$((E_1)(E_2))(E_3) \approx (E_1)\,((E_2)(E_3))$
$(E)(\varepsilon) \approx (\varepsilon)(E) \approx E$

$(E_1)((E_2)|(E_3)) \approx ((E_1)(E_2))|((E_1)(E_3))$
……

Si osservi che, se un'espressione E non contiene nessun carattere di Σ, allora E è equivalente o all'espressione $\varnothing$ oppure all'espressione ε (= $(\varnothing)^*$).

Per evitare una proliferazione di parentesi, ricorriamo all'associatività a sinistra delle tre operazioni (cioè, un oprando con due segni di operazione, uno a sinistra e uno a destra, è associato a quello di sinistra) e a regole di precedenza che stabiliscono che l'operatore di Kleene ha una precedenza più alta della moltiplicazione che a sua volta ha una precedenza più alta dell'unione.

7.3 Linguaggi di Kleene e linguaggi regolari

Proveremo in questo paragrafo che la classe dei linguaggi di Kleene coincide con la classe dei linguaggi regolari. A tale scopo, abbiamo bisogno di alcuni lemmi preliminari.

Lemma 7.1 (*Lemma di Arden*) Data l'equazione

$$\lambda = A \cdot \lambda + B$$

nell'incognita λ dove A ed B sono linguaggi dati,

(*i*) l'insieme delle soluzioni dell'equazione è chiuso rispetto all'unione;

(*ii*) il linguaggio $A^* \cdot B$ è una soluzione dell'equazione ed è contenuto in ogni soluzione dell'equazione;

(*iii*) se A non contiene la stringa vuota, allora il linguaggio $A^* \cdot B$ è l'unica soluzione dell'equazione.

Dimostrazione. (*i*) Siano L_1 ed L_2 due soluzioni dell'equazione cosicché

$$L_1 = A \cdot L_1 + B \qquad\qquad L_2 = A \cdot L_2 + B\ .$$

Allora il linguaggio $L_1 + L_2$ è anche una soluzione dell'equazione visto che

$$L_1 + L_2 = (A \cdot L_1 + B) + (A \cdot L_2 + B) = A \cdot (L_1 + L_2) + B.$$

Dunque, l'insieme delle soluzioni dell'equazione è chiuso rispetto all'unione.

(*ii*) Il linguaggio $A^* \cdot B$ è una soluzione dell'equazione; infatti

$$A^* \cdot B = (A^+ + \{\varepsilon\}) \cdot B = A^+ \cdot B + \{\varepsilon\} \cdot B = A \cdot (A^* \cdot B) + B$$

Dimostriamo ora che il linguaggio $A^* \cdot B$ è contenuto in ogni soluzione dell'equazione. Sia ora L una qualsiasi soluzione dell'equazione. Innanzitutto, si osservi che, siccome $L = A \cdot L + B$, abbiamo che

$$A \cdot L + B \subseteq L,$$

e quindi che

$$A \cdot L \subseteq L \qquad \text{e} \qquad B \subseteq L\ .$$

Per dimostrare che $A^* \cdot B \subseteq L$, è sufficiente provare che $A^n \cdot B \subseteq L$ per ogni $n \geq 0$. Procediamo per induzione.

PASSO BASE. Per $n = 0$, $A^n = \{\varepsilon\}$ e la proprietà segue dall'inclusione $B \subseteq L$.

PASSO INDUTTIVO. Assumiamo che la proprietà sia vera per $n \geq 0$ e dimostriamola per n+1. Siccome

$$A^{n+1} \cdot B = A \cdot A^n \cdot B$$

e per ipotesi $A^n \cdot B \subseteq L$ abbiamo

$$A^{n+1} \cdot B = A \cdot (A^n \cdot B) \subseteq A \cdot L$$

e, siccome $A \cdot L \subseteq L$,

$$A^{n+1} \cdot B = A \cdot (A^n \cdot B) \subseteq A \cdot L \subseteq L.$$

Provata l'inclusione $A^n \cdot B \subseteq L$ per ogni $n \geq 0$, abbiamo allora

$$A^* \cdot B = (\sum_{n=0,\ldots,\infty} A^n) \cdot L = \sum_{n=0,\ldots,\infty} (A^n \cdot L) \subseteq L \ .$$

Dunque, il linguaggio $A^* \cdot B$ è contenuto in ogni soluzione dell'equazione.

(*iii*) Basta provare che l'equazione ha un'unica soluzione se A non contiene la stringa vuota. Sia L la soluzione data dall'unione di tutte le soluzioni dell'equazione e sia L' una qualsiasi soluzione:

$$L = A \cdot L + B \qquad\qquad L' = A \cdot L' + B \ .$$

Sia $C = L \backslash L'$; siccome L' è un sottoinsieme di L, abbiamo $L = L' + C$ e quindi

$$L' + C = A \cdot (L' + C) + B$$

e, per la proprietà distributiva,

$$A \cdot (L' + C) + B = A \cdot L' + A \cdot C + B = A \cdot L' + B + A \cdot C = L' + A \cdot C$$

Dunque

$$L' + C = L' + A \cdot C \ .$$

A questo punto, prendiamo l'intersezione con C di ambo i membri. Siccome $L' \cap C = \varnothing$, otteniamo

$$C = A \cdot C \cap C$$

e, quindi, $C \subseteq A \cdot C$. L'eguaglianza vale banalmente se C è l'insieme vuoto. Dimostriamo ora che C non può che essere vuoto. Supponiamo per assurdo che $C \neq \varnothing$ e sia q, $q \geq 0$, la lunghezza minima delle stringhe appartenenti a C. D'altra parte, se $C \neq \varnothing$ allora anche $A \neq \varnothing$ perché, altrimenti, $C = A \cdot \varnothing \cap C = \varnothing$. Sia m lunghezza minima delle stringhe appartenenti ad A. Ora, siccome $A \neq \varnothing$ e, per ipotesi, $\varepsilon \notin A$, abbiamo $m \geq 1$. Ne segue che la lunghezza minima delle stringhe appartenenti ad $A \cdot C$ è $m+q$ che, siccome $m \geq 1$, è strettamente maggiore di q. Ma, l'inclusione $C \subseteq A \cdot C$ afferma esattamente il contrario e cioè che la lunghezza minima $m+q$ delle stringhe appartenenti ad $A \cdot C$ è minore o uguale alla lunghezza minima q delle stringhe appartenenti a C. Dunque, C deve essere vuoto cosicché deve sempre aversi $L = L'$, cosa che prova l'unicità della soluzione, che è data dal linguaggio $A^* \cdot B$. □

Corollario 7.1 Se A e B sono linguaggi di Kleene ed A non contiene la stringa vuota, allora la soluzione dell'equazione

$$\lambda = A \cdot \lambda + B$$

è un linguaggio di Kleene.

Il Corollario 7.1 consente di provare che ogni linguaggio accettato da un AFD (e quindi ogni linguaggio regolare) è un linguaggio di Kleene. Consideriamo un AFD $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$. Sia $Q = \{q_0, q_1, \ldots, q_n\}$. Per ogni i $(0 \le i \le n)$, indichiamo con λ_i il linguaggio accettato dall'AFD $\boldsymbol{A}_i = (Q, \Sigma, \delta, q_i, F)$. Così $L(\boldsymbol{A}) = L(\boldsymbol{A}_0) = \lambda_0$. Su λ_i possiamo osservare che:

— se x appartiene al linguaggio λ_i e (q_i, a, q_i) è una transizione di $\boldsymbol{A}$, cioè $q_i = \delta(q_i, a)$, allora anche la stringa ax appartiene a λ_i cosicché λ_i include il linguaggio $\{a\} \cdot \lambda_i$;

— se (q_i, a, q_j) è una transizione di $\boldsymbol{A}$, cioè $q_j = \delta(q_i, a)$, ed x appartiene al linguaggio λ_j, allora anche la stringa ax appartiene a λ_i cosicché λ_i include il linguaggio $\{a\} \cdot \lambda_j$;

— se q_i è uno stato di accettazione ($q_i \notin F$), allora la stringa vuota appartiene a λ_i.

Indichiamo con A_{ij} il sottoinsieme (eventualmente vuoto) di Σ tale che a appartiene ad A_{ij} se e solo se (q_i, a, q_j) è una transizione di $\boldsymbol{A}$, cioè $A_{ij} = \{a: q_j = \delta(q_i, a)\}$. Ovviamente, siccome $\boldsymbol{A}$ è un AFD, si ha che $A_{ij} \cap A_{ij'} = \varnothing$ per $j \ne j'$. Poniamo inoltre $B_i = \varnothing$ se $q_i \notin F$ e $B_i = \{\varepsilon\}$ se $q_i \in F$. Si osservi che sia A_{ij} che B_i sono linguaggi di Kleene. Consideriamo ora il sistema formato dalle n+1 equazioni

$$\lambda_i = \sum_j A_{ij} \cdot \lambda_j + B_i \qquad (0 \le i \le n)$$

nelle n+1 incognite $\lambda_0, \lambda_1, \ldots, \lambda_n$. Se sapessimo risolverlo, avremmo anche il valore $L(\boldsymbol{D})$ di λ_0. A tale scopo riscriviamo l'i-esima equazione nella forma

$$\lambda_i = A_{ii} \cdot \lambda_i + (\sum_{j \ne i} A_{ij} \cdot \lambda_j + B_i)$$

Siccome nessun A_{ii} contiene la stringa vuota, per il Lemma di Arden la soluzione dell'i-esima equazione può essere espressa come

$$\lambda_i = A_{ii}^* \cdot (\sum_{j \ne i} A_{ij} \cdot \lambda_j + B_i)$$

e, tenuto conto della proprietà distributiva, come

$$\lambda_i = (\sum_{j \ne i} A_{ii}^* \cdot A_{ij} \cdot \lambda_j) + A_{ii}^* \cdot B_i$$

Si noti che sia i "coefficienti" $A_{ii}^* \cdot A_{ij}$ delle λ_j che la "costante additiva" $A_{ii}^* \cdot B_i$ sono linguaggi di Kleene, e che nessun coefficiente $A_{ii}^* \cdot A_{ij}$ contiene la stringa vuota. Inoltre, se $A_{ii} = \varnothing$, allora $A_{ii}^* = \{\varepsilon\}$ cosicché questa equazione si riduce a

$$\lambda_i = (\sum_{j \ne i} A_{ij} \cdot \lambda_j) + B_i$$

Supponiamo ora di risolvere per prima l'(n+1)-esima equazione

$$\lambda_n = A_{nn} \cdot \lambda_n + (\sum_{j<n} A_{nj} \cdot \lambda_j) + B_n$$

ed otteniamo

$$\lambda_n = A_{nn}^* \cdot ((\sum_{j<n} A_{nj} \cdot \lambda_j) + B_n) = A_{nn}^* \cdot (\sum_{j<n} A_{nj} \cdot \lambda_j) + A_{nn}^* \cdot B_n$$

A questo punto, sostituiamo questa espressione di λ_n nelle restanti n equazioni. Allora, se riscriviamo l'i-esima equazione ($0 \le i \le n-1$) come

$$\lambda_i = A_{ii} \cdot \lambda_i + A_{in} \cdot \lambda_n + (\sum_{j \ne i,n} A_{ij} \cdot \lambda_j) + B_i,$$

e vi sostituiamo l'espressione di λ_n posta nella forma

$$\lambda_n = A_{nn}^* \cdot A_{ni} \cdot \lambda_i + (\sum_{j \ne i,n} A_{nn}^* \cdot A_{nj} \cdot \lambda_j) + A_{nn}^* \cdot B_n$$

otteniamo

$$\begin{aligned} \lambda_i = {} & A_{ii} \cdot \lambda_i + A_{in} \cdot A_{nn}^* \cdot A_{ni} \cdot \lambda_i + \\ & + (\sum_{j \ne i,n} A_{in} \cdot A_{nn}^* \cdot A_{nj} \cdot \lambda_j) + A_{in} \cdot A_{nn}^* \cdot B_n + \\ & + (\sum_{j \ne i,n} A_{ij} \cdot \lambda_j) + B_i = \\ = {} & (A_{ii} + A_{in} \cdot A_{nn}^* \cdot A_{ni}) \cdot \lambda_i + \\ & + (\sum_{j \ne i,n} A_{in} \cdot A_{nn}^* \cdot A_{nj} + A_{ij}) \cdot \lambda_j + A_{in} \cdot A_{nn}^* \cdot B_n + B_i \,. \end{aligned}$$

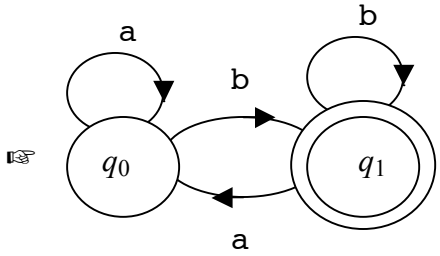
Di nuovo, sia il coefficiente di λ_i sia i coefficienti dei λ_j sia la costante additiva sono linguaggi di Kleene, e nessuno dei coefficienti contiene la stringa vuota. Quindi risolviamo l'(n–1)-esima equazione e così via via, per sostituzione, fino a risolvere la prima equazione che è della forma

$$\lambda_0 = A \cdot \lambda_0 + B$$

dove A e B sono linguaggi di Kleene ed A non contiene la stringa vuota. Pertanto, Per il Lemma 7.1 l'equazione ammette un'unica soluzione, che per il Corollario 7.1 è un linguaggio di Kleene. Abbiamo così dimostrato il seguente risultato.

Lemma 7.2 Il linguaggio accettato da ogni AFD è un linguaggio di Kleene.

Esempio 7.1 Consideriamo l'AFD ***A*** definito dal seguente diagramma delle transizioni:



Ad esso resta associato il sistema delle due equazioni

$$\lambda_0 = \{\mathtt{a}\} \cdot \lambda_0 + \{\mathtt{b}\} \cdot \lambda_1 \qquad\qquad \lambda_1 = \{\mathtt{b}\} \cdot \lambda_1 + \{\mathtt{a}\} \cdot \lambda_0 + \{\varepsilon\}.$$

Per il Lemma di Arden, λ_1 ha la seguente espressione in funzione di λ_0:

$$\lambda_1 = \{\mathtt{b}\}^* \cdot (\{\mathtt{a}\} \cdot \lambda_0 + \{\varepsilon\}) = \{\mathtt{b}\}^* \cdot \{\mathtt{a}\} \cdot \lambda_0 + \{\mathtt{b}\}^*$$

Dopo aver sostituito questa espressione di λ_1 nella prima equazione, questa diventa

$$\begin{aligned}\lambda_0 &= \{\mathtt{a}\} \cdot \lambda_0 + \{\mathtt{b}\}^+ \cdot \{\mathtt{a}\} \cdot \lambda_0 + \{\mathtt{b}\}^+ = \\ &= (\{\mathtt{a}\} + \{\mathtt{b}\}^+ \cdot \{\mathtt{a}\}) \cdot \lambda_0 + \{\mathtt{b}\}^+ \\ &= (\{\varepsilon\} + \{\mathtt{b}\}^+) \cdot \{\mathtt{a}\} \cdot \lambda_0 + \{\mathtt{b}\}^+ \\ &= \{\mathtt{b}\}^* \cdot \{\mathtt{a}\} \cdot \lambda_0 + \{\mathtt{b}\}^+\end{aligned}$$

Per il Lemma di Arden, la soluzione di questa equazione è il linguaggio di Kleene

$$\lambda_0 = (\{\mathtt{b}\}^* \cdot \{\mathtt{a}\})^* \cdot \{\mathtt{b}\}^+.$$

Quindi il linguaggio accettato da ***A*** può essere denotato dall'espressione $(((\mathtt{b})^*(\mathtt{a}))^*)(\mathtt{b})^+$. Possiamo ottenere un'espressione più semplice sfruttando le due identità: (v. sopra)

$$L^+ = L^* \cdot L \qquad\qquad (L^* \cdot M)^* \cdot L^* = (L + M)^* .$$

Così otteniamo l'espressione $((\mathtt{a})|(\mathtt{b}))^*(\mathtt{b})$. ■

Il risultato che segue è conseguenza del Corollario 6.3 e del Lemma 7.2.

Corollario 7.2 Ogni linguaggio regolare è un linguaggio di Kleene.

Ci proponiamo ora di provare che ogni linguaggio di Kleene è un linguaggio regolare. A questo scopo, dimostreremo che, per ogni espressione *E* nell'algebra di Kleene, esiste un ε-AFN che accetta il linguaggio denotato da *E*. L'ε-AFN viene costruito

esplicitamente applicando un insieme di regole all'espressione E; il risultato sarà un un ε-AFN $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$ che ha le seguenti proprietà:

— il numero degli stati di $\boldsymbol{A}$ è minore o uguale al doppio della lunghezza dell'espressione E, cioè $|Q| \leq 2\,|E|$;

— $\boldsymbol{A}$ ha al più uno stato di accettazione, cioè $|F| \leq 1$;

— per ogni stato q di $\boldsymbol{A}$ che non sia di accettazione, abbiamo che

se $\delta(q, \varepsilon) \neq \varnothing$ allora si ha:

$$|\delta(q, \varepsilon)| \leq 2 \qquad \text{e} \qquad \delta(q, a) = \varnothing \text{ per ogni } a \in \Sigma,$$

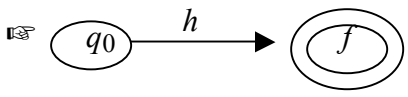
altrimenti esiste un solo carattere a di Σ con $\delta(q, a) \neq \varnothing$ e, in tal caso, $|\delta(q, a)| = 1$.

— non ci sono transizioni di $\boldsymbol{A}$ che abbiano come stato iniziale lo stato di accettazione.

L'automa $\boldsymbol{A}$ prende il nome di *automa di Thompson* associato a E. Vediamo ora come viene costruito. Distinguiamo innanzitutto due casi a seconda che E sia o meno equivalente all'espressione $\varnothing$, cioè a seconda che il linguaggio denotato da E sia o meno vuoto. Se E è equivalente all'espressione $\varnothing$, allora $Q = \{q_0\}$, $F = \varnothing$ e non ci sono transizioni. Se E non è equivalente all'espressione $\varnothing$, allora $\boldsymbol{A}$ avrà sempre uno ed un solo stato di accettazione, che indichiamo con f, cioè $F = \{f\}$. Senza perdere in generalità possiamo assumere che E non contenga nessuna occorrenza di $\varnothing$ anche se possa contenere zero o più occorrenze di ε (= $(\varnothing)^*$).

Regole per la costruzione dell'automa di Thompson

1. Se $E = h$ dove $h \in \Sigma \cup \{\varepsilon\}$, allora $Q = \{q_0, f\}$ e (q_0, h, f) è l'unica transizione di $\boldsymbol{A}$:

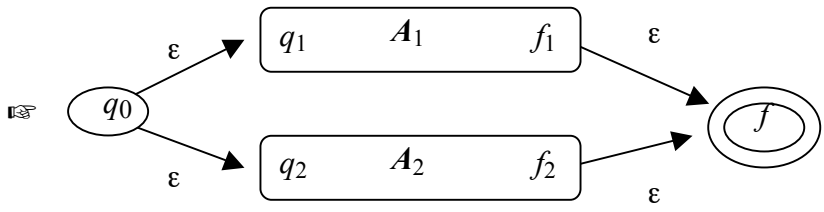


2. Siano $\boldsymbol{A}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1 = \{f_1\})$ e $\boldsymbol{A}_2 = (Q_2, \Sigma, \delta_2, q_2, F_2 = \{f_2\})$ due ε-AFN che accettano rispettivamente i linguaggi denotati dalle espressioni E_1 e E_2. Senza perdere in generalità, assumiamo che Q_1 e Q_2 siano disgiunti.

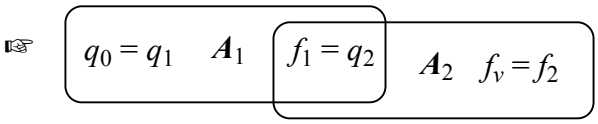
a) Se E è della forma (E_1), allora $\boldsymbol{A} = \boldsymbol{A}_1$.

b) Se E è della forma $(E_1)|(E_2)$, allora $Q = \{q_0, f\} \cup Q_1 \cup Q_2$ e le transizioni di $\boldsymbol{A}$ sono le transizioni di $\boldsymbol{A}_1$ e di $\boldsymbol{A}_2$ più le quattro transizioni spontanee

(q_0, ε, q_1) (q_0, ε, q_2) (f_1, ε, f) (f_2, ε, f).

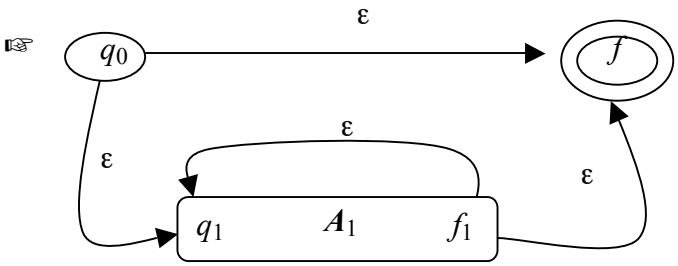


c) Se E è della forma $(E_1)(E_2)$, allora $Q = Q_1 \cup Q_2$, $q_0 = q_1$, $f = f_2$, gli stati f_1 e q_2 sono fusi in unico stato e le transizioni di $\boldsymbol{A}$ sono le transizioni di $\boldsymbol{A}_1$ e di $\boldsymbol{A}_2$. finito



d) Se E è della forma $(E_1)^*$, allora $Q = \{q_0, f\} \cup Q_1$ e le transizioni di $\boldsymbol{A}$ sono le transizioni di $\boldsymbol{A}_1$ più le quattro transizioni spontanee

(q_0, ε, q_1) (q_0, ε, f) (f_1, ε, q_1) (f_1, ε, f).



Si lascia come esercizio la dimostrazione del seguente risultato.

Lemma 7.3 Per ogni espressione E nell'algebra di Kleene, l'automa di Thompson associato a E accetta il linguaggio denotato da E.

Combinando il Corollario 7.2 ed il Lemma 7.3, abbiamo il risultato seguente.

Teorema 7.2 (*Teorema di Kleene*) La classe dei linguaggi di Kleene coincide con la classe dei linguaggi regolari.

Alla luce del Teorema di Kleene, le espressioni nell'algebra di Kleene sono chiamate *espressioni regolari*.

7.4 Algoritmo di McNaughton-Yamada-Thompson

Diamo ora un algoritmo che esplicitamente costruisce l'automa di Thompson associato ad un'espressione dell'algebra di Kleene. Il punto di partenza è la seguente grammatica generativa delle espressioni regolari, che senza perdere in generalità supponiamo prive del simbolo $\varnothing$:

$$\boldsymbol{G} = (\{S\}, \Omega, S, P)$$

dove $\Omega = \Sigma \cup \{\varepsilon, |, *, (,)\}$ e P contiene le $5+|\Sigma|$ produzioni:

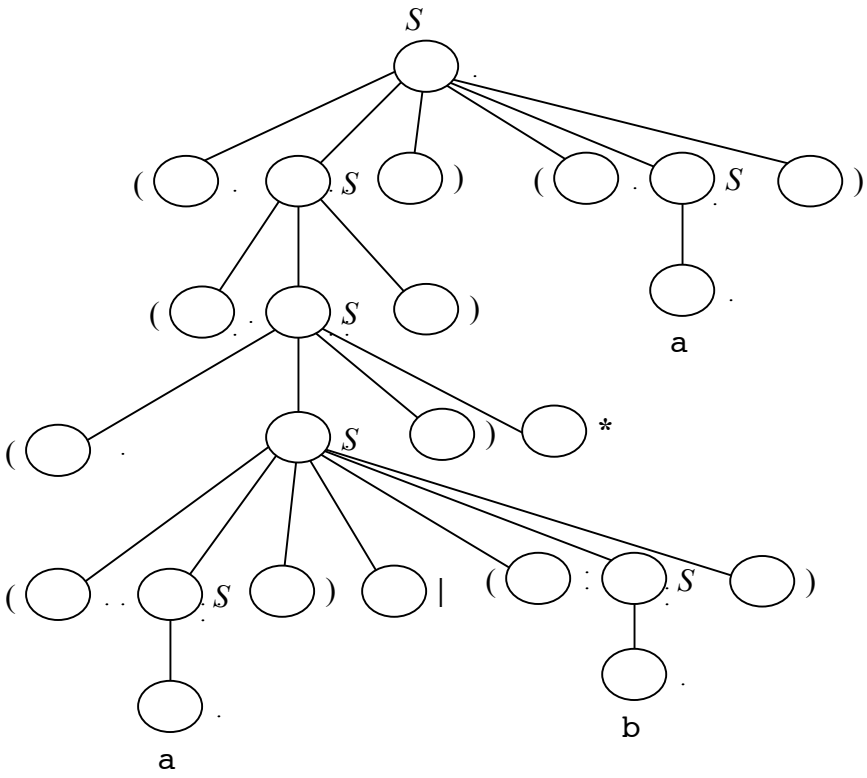
$$\begin{aligned}
&S \to (S)\\
&S \to (S)|(S)\\
&S \to (S)(S)\\
&S \to (S)^*\\
&S \to a \qquad \text{per ogni carattere } a \in \Sigma\\
&S \to \varepsilon
\end{aligned}$$

Grazie all'uso (forse eccessivo) delle parentesi, $\boldsymbol{G}$ non è una grammatica ambigua; inoltre, una stringa su Ω è una frase di $\boldsymbol{G}$ se e solo se è un'espressione dell'algebra di Kleene su Σ. Sia ora E una frase di $\boldsymbol{G}$ e sia $\mathcal{T}$ l'albero di derivazione per E. Consideriamo un qualsiasi vertice v di $\mathcal{T}$ e sia $\mathcal{T}_v$ il sottoalbero di $\mathcal{T}$ che ha per radice il vertice v. La concatenazione delle foglie di $\mathcal{T}_v$ è una sottostringa di E ed è anch'essa una frase di $\boldsymbol{G}$, che chiamiamo la *sottoespressione* E_v di E.
Con una visita di $\mathcal{T}$ che proceda dalle foglie alla radice, per ogni vertice interno v di $\mathcal{T}$ costruiamo l'automa di Thompson associato all'espressione E_v. Terminata la visita di $\mathcal{T}$, l'automa di Thompson associato alla radice di $\mathcal{T}$ fornisce proprio l'automa di Thompson associato a E.

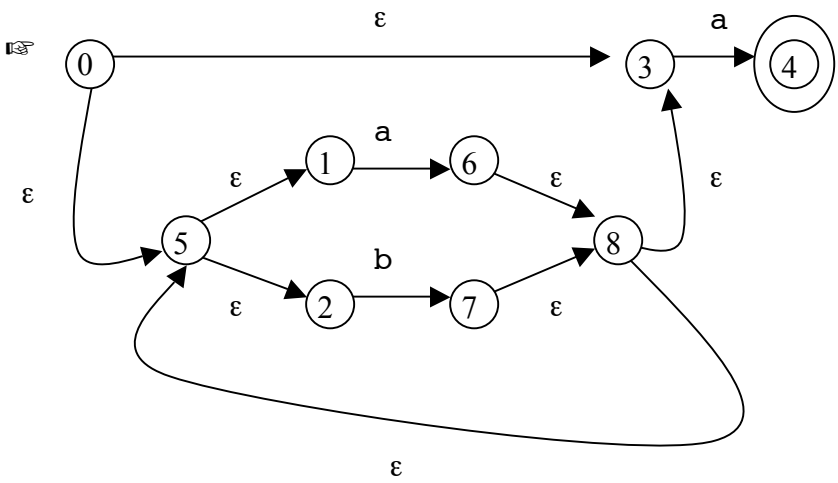
Esempio 7.2 Consideriamo l'espressione

$$E = (((\texttt{a})|(\texttt{b}))^*)(\texttt{a}).$$

Il linguaggio denotato da E è $\{\texttt{a}, \texttt{b}\}^* \cdot \{\texttt{a}\}$. L'albero di derivazione $\mathcal{T}$ di E è mostrato in figura.



L'automa di Thompson ***A*** associato all'espressione *E* è mostrato in figura.



7.5 Automa di Thompson ridotto

Sia E un'espressione regolare che non è equivalente né a $\varnothing$ né a ε. L'automa di Thompson $\boldsymbol{A}$ associato ad E può facilmente essere semplificato. A tal fine, chiamiamo *importanti* gli stati iniziali di transizioni nonspontanee, nonché lo stato iniziale di $\boldsymbol{A}$ e lo stato di accettazione di $\boldsymbol{A}$; inoltre, diciamo che una sequenza di transizioni è importante se lo stato iniziale e lo stato finale della sequenza sono stati importanti e se nessuno degli stati intermedi è importante. Sia Q^* l'insieme degli stati importanti di $\boldsymbol{A}$. Sia $\boldsymbol{A}^*$ l'ε-AFN su Q^* che si ottiene da $\boldsymbol{A}$ condensando ogni sequenza importante di transizioni tra stati in Q^* in un'unica transizione etichettata dall'etichetta (carattere o ε) della prima transizione della sequenza. È facile convincersi che $\boldsymbol{A}^*$ è equivalente ad $\boldsymbol{A}$. Chiamiamo $\boldsymbol{A}^*$ l'*automa di Thompson ridotto* associato a E.

Esempio 7.2 (seguito) Gli stati importanti di $\boldsymbol{A}$ sono: 0, 1, 2, 3 e 4. Le sequenze importanti di transizioni di $\boldsymbol{A}$ sono:

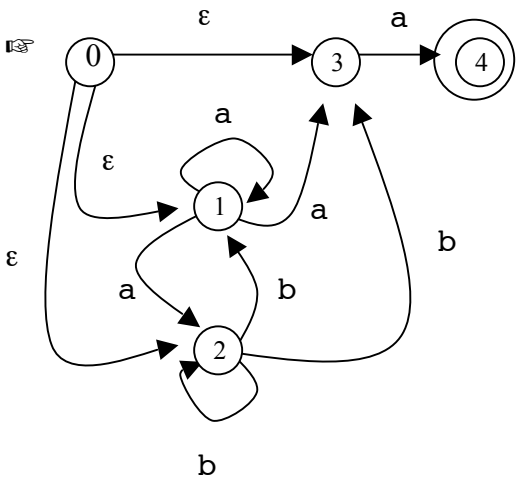
(0, ε, 3) (0, ε, 5, ε, 1) (0, ε, 5, ε, 2)

(1, a, 6, ε, 8, ε, 5, ε, 1) (1, a, 6, ε, 8, ε, 5, ε, 2) (1, a, 6, ε, 8, ε, 3)

(2, b, 7, ε, 8, ε, 5, ε, 1) (2, b, 7, ε, 8, ε, 5, ε, 2) (2, b, 7, ε, 8, ε, 3)

(3, a, 4)

L'automa di Thompson ridotto $\boldsymbol{A}^*$ associato a E è mostrato in figura.



■

A partire dall'automa di Thomson ridotto $\boldsymbol{A}^*$, possiamo costruire un AFD equivalente ad $\boldsymbol{A}^*$ (e quindi ad $\boldsymbol{A}$) (e poi magari uno di dimensione minima) usando l'Algoritmo

6.2. Ma, grazie alla semplice struttura di $\boldsymbol{A}^*$, è più conveniente usare una variante dell'Algoritmo 6.2, basata sul concetto di "adiacenza". Uno stato q' di $\boldsymbol{A}^*$ è *adiacente* ad un altro stato q *rispetto* a un carattere a se (q, a, q') è una transizione di $\boldsymbol{A}^*$; con $N_a(q)$ indichiamo l'insieme degli stati adiacenti a q rispetto ad a.

Algoritmo 7.2

1. $Q_{\boldsymbol{D}} := \{[q_0]_\varepsilon\}$, dove q_0 è lo stato iniziale di $\boldsymbol{A}^*$.

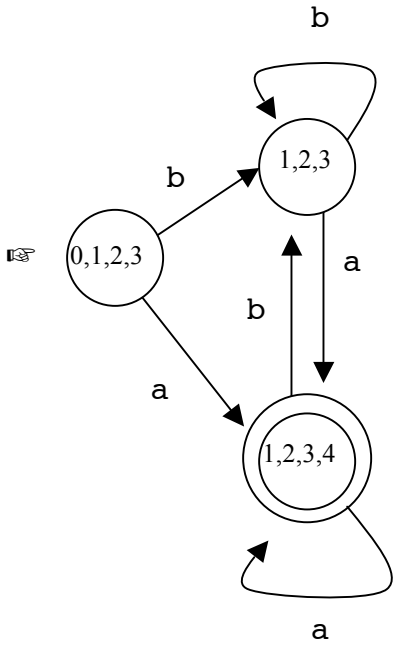
2. Dichiarare lo stato $[q_0]_\varepsilon$ di $\boldsymbol{D}$ "aperto".

3. Fintantoché esiste uno stato aperto $q_{\boldsymbol{D}}$ di $\boldsymbol{D}$, ripetere

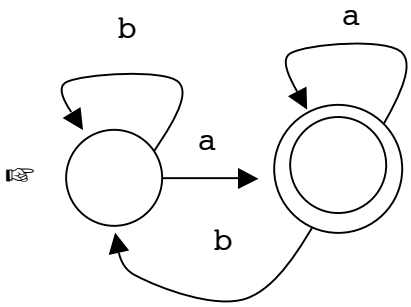
    dichiarare $q_{\boldsymbol{D}}$ "chiuso";
    per ogni $a \in \Sigma$, ripetere
        $p := \bigcup_{q \in q_{\boldsymbol{D}}} N_a(q)$;
        se $p \notin Q_{\boldsymbol{D}}$, allora
            aggiungere p a $Q_{\boldsymbol{D}}$;
            dichiarare p "aperto";
        $\delta_{\boldsymbol{D}}(q_{\boldsymbol{D}}, a) := p$.

3. $F_{\boldsymbol{D}} := \{q_{\boldsymbol{D}} \in Q_{\boldsymbol{D}}: q_{\boldsymbol{D}} \cap F \neq \varnothing\}$.

Esempio 7.2 (seguito) Quando applichiamo ad $\boldsymbol{A}^*$ l'Algoritmo 6.2 oppure l'Algoritmo 7.2 otteniamo l'AFD $\boldsymbol{D}$ mostrato in figura.

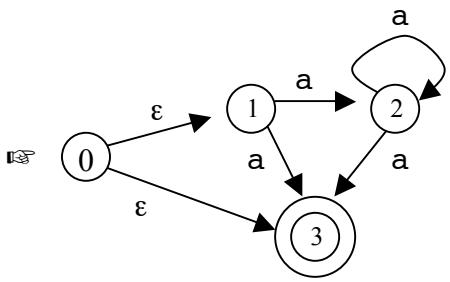


Si osservi che gli stati {0, 1, 2, 3} e {1, 2, 3} sono indistinguibili e l'AFD mostrato in figura

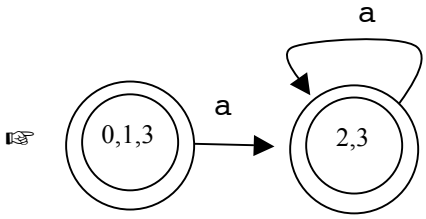


è equivalente a ***D*** ed è di dimensione minima. ■

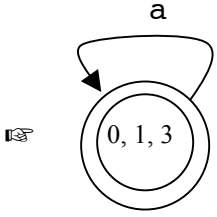
Esempio 7.3 Consideriamo l'espressione E = (((a)((a)*)))|(ε) su Σ = {a}. L'automa di Thompson ridotto associato a E è



Quando applichiamo l'Algoritmo 7.2 all'automa di Thompson ridotto otteniamo l'automa finito ***A*** mostrato in figura.



Un automa finito equivalente ad ***A*** e di dimensione minima è il seguente.



■

7.6 Algoritmi di riconoscimento

Il problema del riconoscimento delle stringhe appartenenti al linguaggio denotato da un'espressione regolare E può essere risolto utilizzando l'automa di Thompson $\boldsymbol{A}$ associato a E. Siccome il numero degli stati di $\boldsymbol{A}$ è $O(|E|)$ e ci sono al più due transizioni da ogni stato di $\boldsymbol{A}$, risolvere il problema del riconoscimento richiede un tempo $O(|E| \cdot |x|)$ se x è la stringa in esame. In alternativa, a partire da E possiamo usare un ADF $\boldsymbol{D}$ di dimensione minima che accetta il linguaggio denotato da E. Ovviamente, se si usa $\boldsymbol{D}$, allora il processo è più semplice, ma il guadagno va commisurato con il costo computazionale della costruzione di $\boldsymbol{D}$ che, come si sa, potrebbe avere un numero esponenziale di stati nella lunghezza di E.

Cap. 8

ANALISI SINTATTICA

Sia $\boldsymbol{G} = (N, T, P, S)$ la grammatica che definisce la sintassi del linguaggio sorgente. L'analisi sintattica ha il compito primario di decidere se lo schema lessicale del testo è o meno una frase di $\boldsymbol{G}$, cioè se il testo è o meno un programma nel linguaggio sorgente. A tale scopo potremmo utilizzare l'algoritmo CYK che, se lo schema lessicale è una frase della grammatica $\boldsymbol{G}$, consente anche di costruire l'albero di derivazione per lo schema lessicale.
Consideriamo le produzioni

$$
\begin{array}{lcl}
I & \rightarrow & \textbf{id ass } E \\
 & & \mid \textbf{if } E \textbf{ then } I \\
 & & \mid \textbf{if } E \textbf{ then } I \textbf{ else } I \\
 & & \mid \textbf{while } E \textbf{ do } I \\
 & & \mid \textbf{begin } I \textbf{ end} \\
E & \rightarrow & S \mid S \textbf{ comp } S \\
S & \rightarrow & T \mid S \textbf{ add } T \\
T & \rightarrow & \textbf{id} \mid \textbf{num} \mid T \textbf{ mol } F \\
F & \rightarrow & \textbf{id} \mid \textbf{num} \mid F \textbf{ exp } F
\end{array}
$$

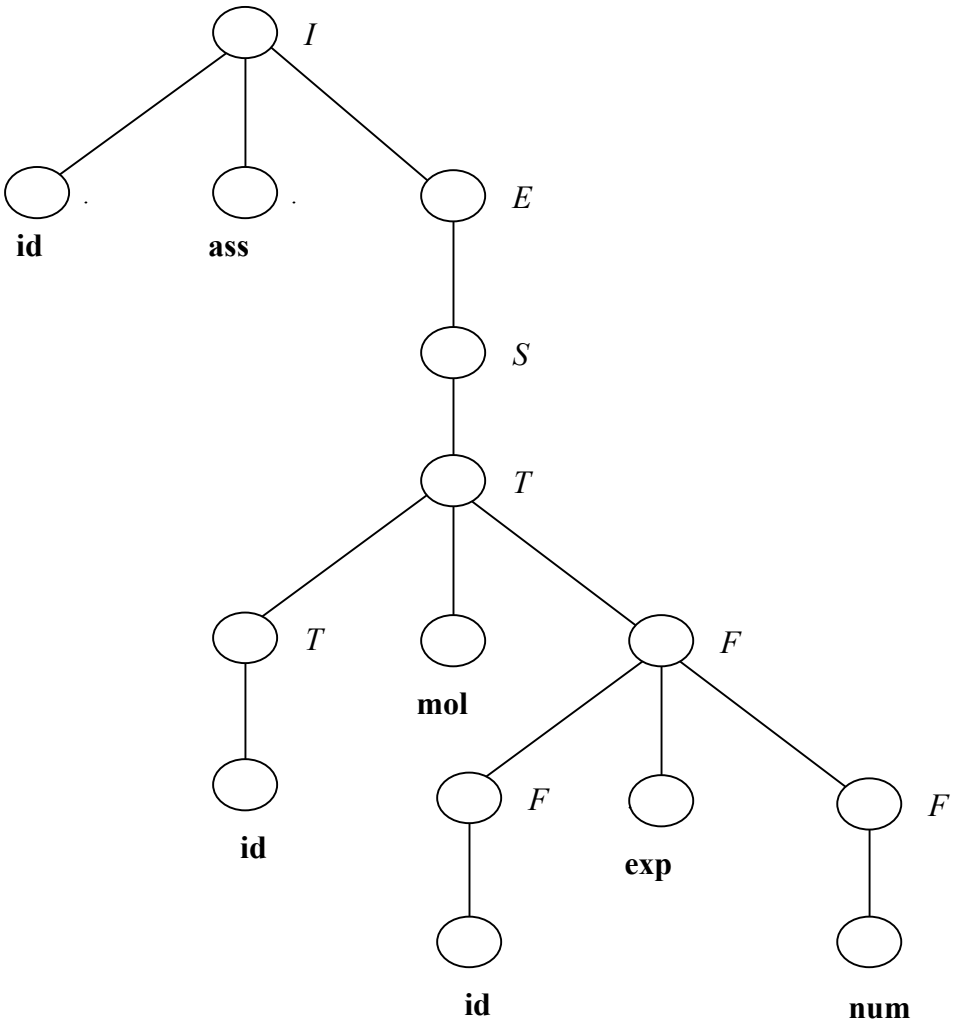
e l'istruzione

```
e:=m*c**2
```

Il suo schema lessicale è

id ass id mol id exp num

ed è derivabile da *I*. Un suo albero di derivazione è mostrato in figura.

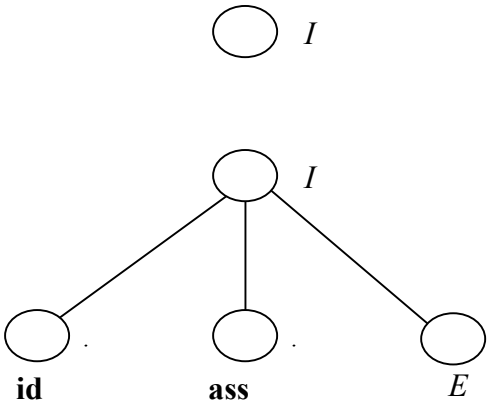


Le derivazioni sinistrorsa e destrorsa sono riportate qui di seguito

derivazione sinistrorsa	*derivazione destrorsa*
I	*I*
id ass *E*	**id ass** *E*
id ass *S*	**id ass** *S*
id ass *T*	**id ass** *T*
id ass *T* **mol** *F*	**id ass** *T* **mol** *F*
id ass id mol *F*	**id ass** *T* **mol** *F* **exp** *F*
id ass id mol *F* **exp** *F*	**id ass** *T* **mol** *F* **exp num**
id ass id mol id exp *F*	**id ass** *T* **mol id exp num**
id ass id mol id exp num	**id ass id mol id exp num**

Per decidere se lo schema lessicale del testo è o meno una frase della grammatica, potremmo utilizzare l'algoritmo CYK la cui complessità però lo rende poco efficiente per essere impiegato nei compilatori. In pratica, si utilizzano metodi più efficienti

(anche se meno generali) che vengono comunemente classificati in *metodi deduttivi* e *metodi induttivi*. I primi provano ad ottenere una derivazione sinistrorsa costruendo un albero di derivazione a partire dalla radice:

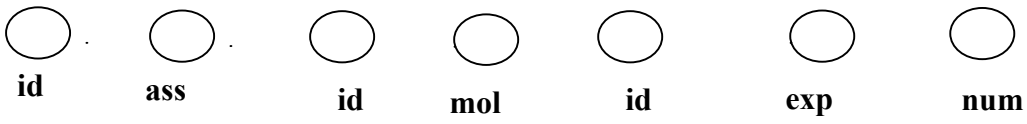


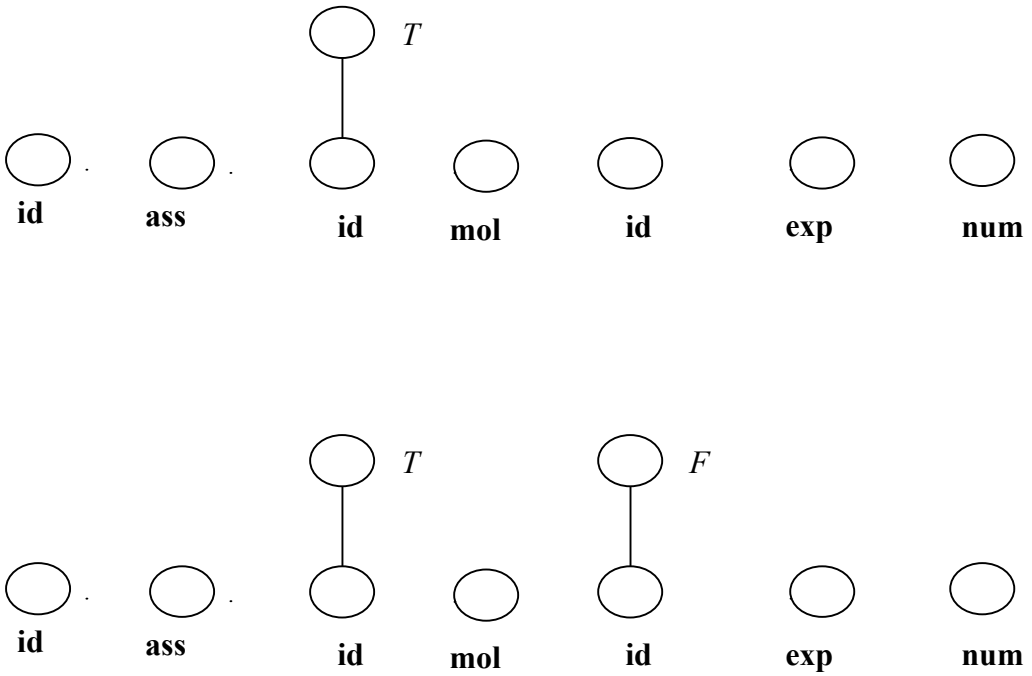
…

I secondi invece provano ad ottenere la sequenza inversa di una derivazione destrorsa

id ass id mol id exp num
id ass *T* **mol id exp num**
id ass *T* **mol** *F* **exp num**
id ass *T* **mol** *F* **exp** *F*
id ass *T* **mol** *F*
id ass *T*
id ass *S*
id ass *E*
I

costruendo un albero di derivazione a partire dalle foglie:





In entrambi i casi, lo schema lessicale (la stringa d'ingresso) è letto dall'inizio alla fine (cioè da sinistra a destra), un simbolo (figura lessicale) per volta.

Per applicare sia il metodo deduttivo che quello induttivo, lo schema lessicale viene completato con il segno (di fine stringa) `$`, viene caricato in un buffer e viene letto utilizzando un puntatore $\triangle$, chiamato il *cursore*, che ne cadenza la lettura. Chiameremo la figura lessicale indicata dal cursore il *simbolo corrente*. Inoltre, entrambi i metodi fanno uso di due funzioni.

La prima funzione, che indichiamo con $\mathcal{I}$ (FIRST) è definita per stringhe di simboli grammaticali (stringhe di figure lessicali e sintattiche):

> $\mathcal{I}(\alpha)$ è l'insieme dei simboli terminali con cui iniziano le stringhe derivabili da α; inoltre, se α è una stringa di sole variabili annullabili, cioè se ε è derivabile da α, allora anche ε è un elemento di $\mathcal{I}(\alpha)$. Pertanto, $\mathcal{I}(\alpha)$ è sempre un sottoinsieme di $T \cup \{\varepsilon\}$.

La seconda funzione, che indichiamo con $\mathcal{J}$ (FOLLOW), è definita per simboli nonterminali:

$\mathcal{J}(A)$ è l'insieme dei simboli terminali che possono trovarsi subito dopo A in una qualsiasi formula della grammatica $\boldsymbol{G}$; cioè, un simbolo terminale a appartiene a $\mathcal{J}(A)$ se e solo se Aa è una sottostringa di una qualche formula di $\boldsymbol{G}$. Inoltre, $\mathcal{J}(A)$ può contenere il segno `$`. Pertanto, $\mathcal{J}(A)$ è sempre un sottoinsieme di $T \cup \{\$\}$.

Per calcolare $\mathcal{I}(\alpha)$ abbiamo bisogno del seguente algoritmo che, una volta determinate le variabili annullabili, costruisce gli insiemi $\mathcal{I}(X)$ per ogni simbolo grammaticale X di $\boldsymbol{G}$.

Algoritmo 8.1

1. Per ogni simbolo terminale a di $\boldsymbol{G}$, porre $\mathcal{I}(a) := \{a\}$, e per ogni simbolo nonterminale A porre $\mathcal{I}(A) =: \varnothing$.

2. Finché per nessun simbolo nonterminale A l'insieme $\mathcal{I}(A)$ possa essere ulteriormente ampliato, ripetere

   per ogni produzione nonnulla $A \rightarrow \alpha$ di $\boldsymbol{G}$, sia β il più lungo prefisso di α che contiene solo variabili annullabili;

   se $\beta \neq \varepsilon$ allora, per ogni simbolo nonterminale B presente in β, porre $\mathcal{I}(A) := \mathcal{I}(A) \cup \mathcal{I}(B)$;

   se $\beta \neq \alpha$ allora, detto X il simbolo grammaticale che occupa la posizione $(|\beta|+1)$.esima in α, porre $\mathcal{I}(A) := \mathcal{I}(A) \cup \mathcal{I}(X)$.

3. Per ogni variabile annullabile A, porre $\mathcal{I}(A) := \mathcal{I}(A) \cup \{\varepsilon\}$.

Una volta calcolato l'insieme $\mathcal{I}(X)$ per ogni simbolo grammaticale X di $\boldsymbol{G}$, con l'algoritmo seguente possiamo ora costruire l'insieme $\mathcal{I}(\alpha)$ per una data stringa α.

Algoritmo 8.2

1. Se $\alpha = \varepsilon$, allora porre $\mathcal{I}(\alpha) := \{\varepsilon\}$; altrimenti, $\mathcal{I}(\alpha) := \varnothing$.

2. Se $\alpha \neq \varepsilon$, allora

   sia β il più lungo prefisso di α che contiene solo variabili annullabili;

   se $\beta \neq \varepsilon$ allora, per ogni simbolo nonterminale B presente in β, porre $\mathcal{I}(\alpha) := \mathcal{I}(\alpha) \cup (\mathcal{I}(B) \setminus \{\varepsilon\})$;

se $\beta \neq \alpha$ allora, detto X il simbolo grammaticale che occupa la posizione ($|\beta|$+1).esima in α, porre $\mathcal{I}(\alpha) := \mathcal{I}(\alpha) \cup \mathcal{I}(X)$.

se $\beta = \alpha$ porre $\mathcal{I}(\alpha) := \mathcal{I}(\alpha) \cup \{\varepsilon\}$; altrimenti, detto X il simbolo grammaticale che occupa la posizione ($|\beta|$+1).esima in α, porre $\mathcal{I}(\alpha) := \mathcal{I}(\alpha) \cup \mathcal{I}(X)$.

Passiamo ora alla funzione $\mathcal{J}$. Il seguente algoritmo costruisce gli insiemi $\mathcal{J}(A)$ per ogni simbolo nonterminale A di $\boldsymbol{G}$ e si basa sulla seguente proprietà:

Per ogni produzione di $\boldsymbol{G}$ della forma $A \rightarrow \beta B \gamma$

(*i*) se $a \in \mathcal{I}(\gamma)$ allora $a \in \mathcal{J}(B)$;

(*ii*) se $a \in \mathcal{J}(A)$ ed $\varepsilon \in \mathcal{I}(\gamma)$ (cioè la stringa vuota è derivabile da γ), allora $a \in \mathcal{J}(B)$.

Algoritmo 8.3

1. Porre $\mathcal{J}(S) := \{\$\}$. Per ogni altro simbolo nonterminale A di $\boldsymbol{G}$, $\mathcal{J}(A) := \varnothing$.

2. Finché per nessun simbolo nonterminale A insieme $\mathcal{J}(A)$ possa essere ulteriormente ampliato, ripetere:

   per ogni produzione $A \rightarrow \alpha$ di $\boldsymbol{G}$ tale che α contenga almeno un simbolo nonterminale

   per ogni simbolo nonterminale B contenuto in α

   per ogni suffisso γ di α preceduto da B (cioè $A \rightarrow \beta B \gamma$)

   calcolare $\mathcal{I}(\gamma)$ con l'Algoritmo 8.2;
   $\mathcal{J}(B) := \mathcal{J}(B) \cup (\mathcal{I}(\gamma) \backslash \{\varepsilon\})$;
   se $\varepsilon \in \mathcal{I}(\gamma)$ allora $\mathcal{J}(B) := \mathcal{J}(B) \cup \mathcal{J}(A)$.

Consideriamo a mo' d'esempio, la seguente grammatica che genera un certo sottoinsieme dei tipi nel linguaggio Pascal:

tipo → *semplice* | ↑ **id** | **array** [*semplice*] **of** *tipo*

semplice → **integer** | **char** | **num . . num**

Allora abbiamo per i simboli grammaticali

X	$\mathcal{I}(X)$
integer	**integer**
char	{**char**}
num	{**num**}
. .	{**. .**}
id	{**id**}
↑	{↑}
array	{**array**}
of	{**of**}
[	{**[**}
]	{**]**}
semplice	{**integer**, **char**, **num**}
tipo	{**integer**, **char**, **num**, ↑, **array**}

e per le stringhe

α	$\mathcal{I}(\alpha)$
ε	{ε}
↑ **id**	{↑}
array **[** *semplice* **]** **of** *tipo*	{**array**}
num . . num	{**num**}
……	……
] **of** *tipo*	{**]**}
……	……

Infine, per le variabili abbiamo

A	$\mathcal{J}(A)$
tipo	{**$**}
semplice	{**$**, **]**}

Si osservi che $\mathcal{J}$(*semplice*) contiene **$** per via della produzione *tipo* → *semplice* e contiene **]** per via della produzione *tipo* → **array** **[** *semplice* **]** **of** *tipo.*

Consideriamo poi le produzioni

E → **E+T** | **T**
T → **T*F** | **F**

$\mathtt{F} \rightarrow \mathtt{(E)} \mid \mathbf{id}$

Quando applichiamo gli Algoritmi 8.1, 8.2 e 8.3 otteniamo

$\mathcal{I}(\mathtt{E}) = \mathcal{I}(\mathtt{T}) = \mathcal{I}(\mathtt{F}) = \{\mathtt{(}, \mathbf{id}\}$

$\mathcal{J}(\mathtt{E}) = \mathcal{J}(\mathtt{T}) = \mathcal{J}(\mathtt{F}) = \{\mathtt{\$}, \mathtt{+}, \mathtt{*}, \mathtt{)}\}$

8.1 Approccio deduttivo

La costruzione di un albero di derivazione per lo schema lessicale inizia con un albero formato da un unico vertice (la radice) etichettato dal simbolo iniziale *S* della grammatica ***G*** e procede visitando i vertici nell'ordine "primo in profondità" e "sviluppando" i vertici etichettati con simboli nonterminali. Il metodo fa uso di un puntatore ↟, chiamato la *sonda*, che sempre indica una foglia dell'albero e segnala lo stato di avanzamento nella costruzione dell'albero di derivazione. In modo equivalente, il metodo cerca di costruire una derivazione sinistrorsa dello schema lessicale.

Consideriamo di nuovo la grammatica dei tipi e lo schema lessicale

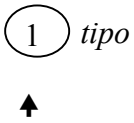
array [num .. num] of integer

Così il buffer contiene la stringa

array [num .. num] of integer $

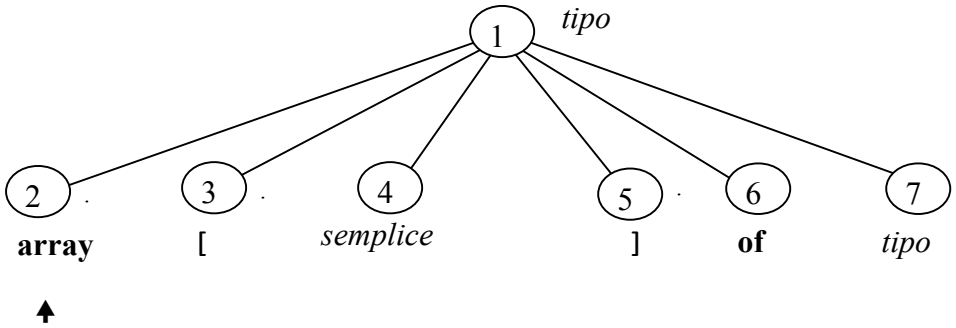
△

e **array** è il simbolo corrente. Inizialmente, l'albero di derivazione ha un solo vertice etichettato con il simbolo iniziale *tipo*:



Così, *tipo* è la prima formula della derivazione sinistrorsa che andiamo a costruire. Ora, il nodo di controllo 1 viene sviluppato apaplicando la produzione

tipo → **array** [*semplice*] **of** *tipo*

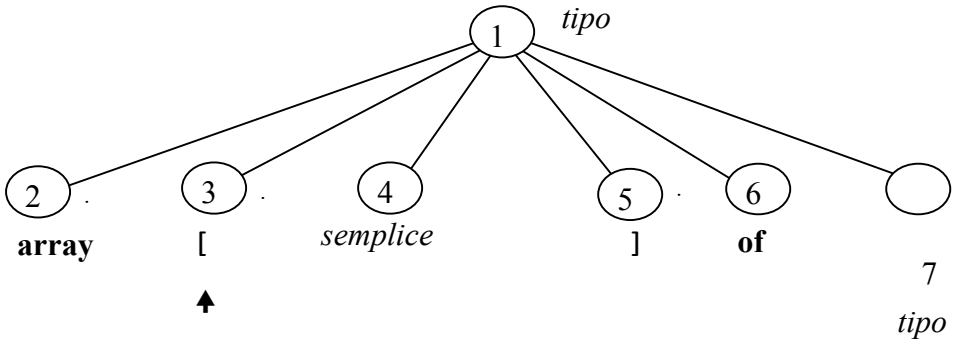


Il segmento di derivazione sinistrorsa che se ne ottiene è

tipo
array [*semplice*] **of** *tipo*.

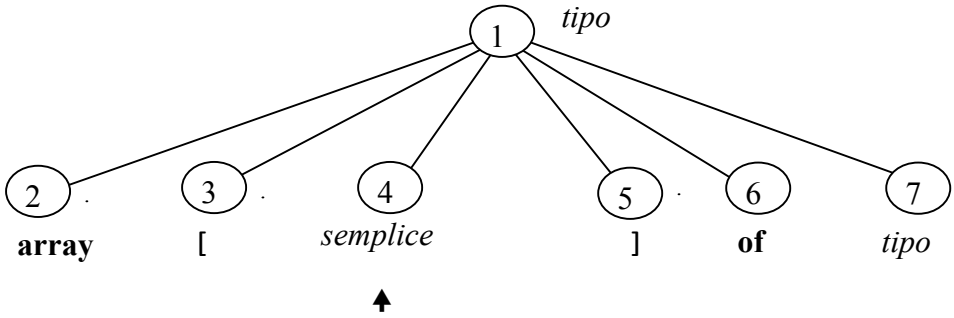
Siccome il nodo di controllo 2 ha per etichetta un simbolo terminale (**array**) e questo coincide con il simbolo corrente della stringa di ingresso, allora il cursore △ e la sonda ↟ vengono entrambi fatti avanzare: il nuovo simbolo corrente è **[** ed il nuovo nodo di controllo è il vertice 3, che è il secondo figlio del vertice 1.
La situazione è quella rappresentata in figura.

array [num .. num] of integer $
△



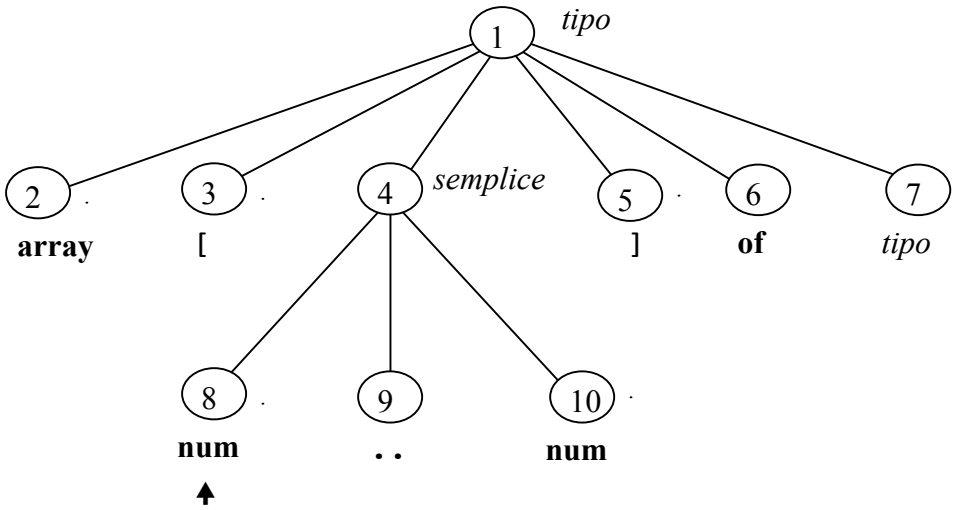
Di nuovo, il nodo di controllo 3 ha per etichetta un simbolo terminale (**[**) e questo coincide con il simbolo corrente cosicché il cursore △ ed la sonda ↟ vengono entrambi fatti avanzare e la situazione ora è quella rappresentata in figura.

array [num .. num] of integer $
△



A questo punto, il nodo di controllo 4 viene sviluppato applicando la produzione

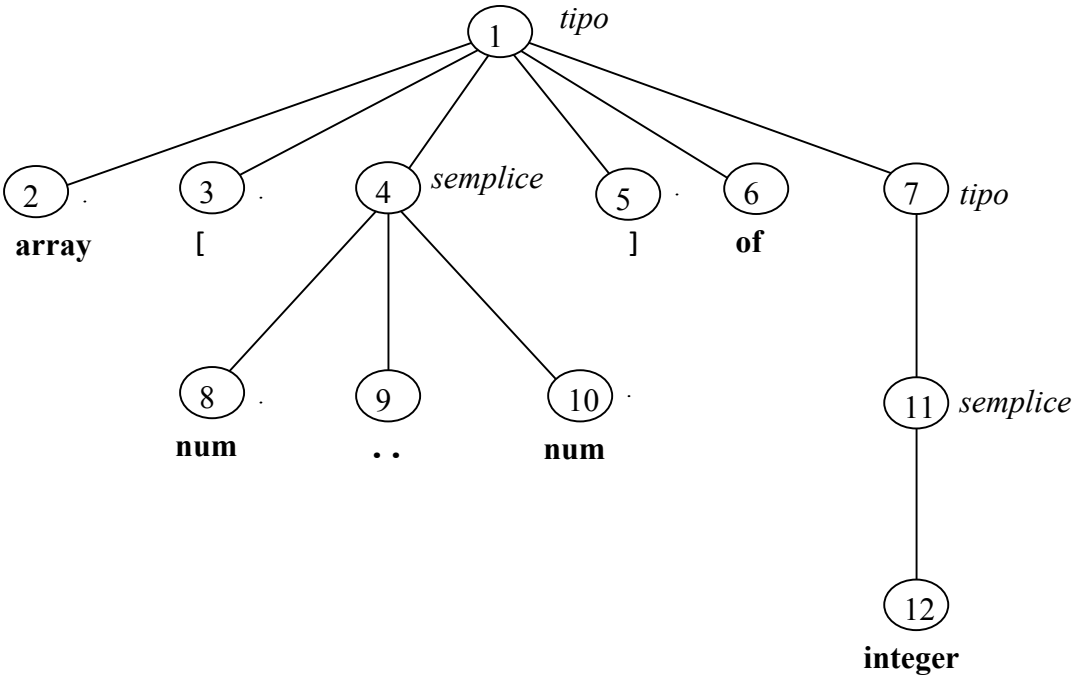
semplice → **num . . num**



Il segmento di derivazione sinistrorsa così ottenuto è

tipo
array [*semplice*] **of** *tipo*
array [**num . . num**] **of** *tipo*.

Procedendo in questa maniera, riusciamo a costruire l'intero albero di derivazione (v. figura) per la stringa di ingresso



e ci accorgiamo che il lavoro è completato perché il simbolo corrente è `$` e non ci sono più vertici da sviluppare.

La derivazione sinistrorsa che si è così ottenuta è

tipo
array `[` *semplice* `]` **of** *tipo*
array `[` **num . . num** `]` **of** *tipo*
array `[` **num . . num** `]` **of** *semplice*
array `[` **num . . num** `]` **of integer**

Vediamo ora il generico passo del metodo. Supponiamo che *a* sia il simbolo corrente, che $a \neq$ `$` e che *u* sia il nodo di controllo.

Distinguiamo i casi seguenti:

Caso 1: l'etichetta di *u* è *a*. Allora, il cursore △ e la sonda ↟ vengono entrambi fatti avanzare: il cursore punterà al simbolo successivo ad *a* nella stringa d'ingresso e la sonda punterà alla foglia successiva a *u*. Se *u* era l'ultima foglia ed il simbolo

successivo ad *a* è $, allora l'algoritmo termina; altrimenti, siamo in un "vicolo cieco" (v. appresso).

Caso 2: l'etichetta di *u* è una figura lessicale diversa da *a*. Allora siamo in un "vicolo cieco" (v. appresso).

Caso 3: l'etichetta di *u* è la stringa vuota. Allora la sonda ↟ risale e punterà alla foglia successiva a *u*. Se non vi sono altre foglie, allora siamo in un "vicolo cieco" (v. appresso).

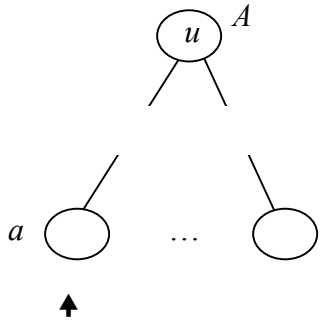
Caso 4: l'etichetta di *u* è una variable sintattica. Allora si sviluppa il nodo *u* ed la sonda ↟ scende al primo figlio di *u*.

Vediamo ora come viene scelta la produzione da applicare per sviluppare il nodo di controllo nel Caso 4. La scelta viene fatta consultando un'apposita tabella, chiamata *tabella di parsing predittivo*, che, per ogni coppia (*A*, *e*) dove *A* è un simbolo nonterminale di ***G*** ed *e* è un elemento di $T \cup \{\$\}$, riporta le produzioni $A \rightarrow \alpha$ della grammatica per le quali

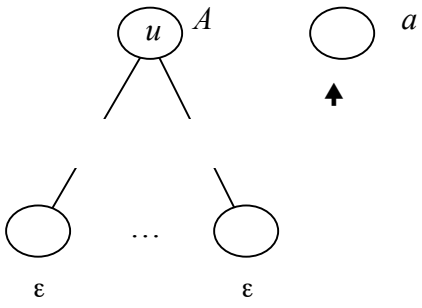
(I) $e \in \mathcal{I}(\alpha)$

(II) $\varepsilon \in \mathcal{I}(\alpha)$ (cioè, α è una stringa di sole variabili annullabili) ed $e \in \mathcal{J}(A)$.

Così, nel Caso 4 si sceglie di applicare una produzione $A \rightarrow \alpha$ tale che la stringa α sia contenuta nella cella (*A*, *a*) della tabella di parsing predittivo. Si osservi che scegliendo $a \in \mathcal{I}(\alpha)$ prefiguriamo per il sottoalbero con radice *u* dell'albero di derivazione la forma seguente



mentre scegliendo $a \in \mathcal{J}(A)$ prefiguriamo per il sottoalbero con radice *u* dell'albero di derivazione la forma seguente



Per costruire la tabella di parsing predittivo, conviene talora costruire preliminarmente una tabella ausiliaria (la *tabella $\mathcal{I}$-$\mathcal{J}$*) che riporti per ogni produzione $A \rightarrow \alpha$ della grammatica l'insieme $\mathcal{I}(\alpha)$ e, se $\varepsilon \in \mathcal{I}(\alpha)$, anche l'insieme $\mathcal{J}(A)$. A questo punto, per ottenere la tabella di parsing predittivo si applica il seguente algoritmo:

Per ogni produzione $A \rightarrow \alpha$ della grammatica,

per ogni simbolo terminale *a* in $\mathcal{I}(\alpha)$, inserire $A \rightarrow \alpha$ nella cella (A, a);

se $\varepsilon \in \mathcal{I}(\alpha)$, allora

per ogni elemento *e* di $\mathcal{J}(A)$, inserire $A \rightarrow \alpha$ nella cella (A, e).

Per la nostra grammatica dei tipi abbiamo la tabella $\mathcal{I}$-$\mathcal{J}$

$A \rightarrow \alpha$	$\mathcal{I}(\alpha)$	$\mathcal{J}(A)$
tipo → *semplice*	{**integer**, **char**, **num**}	
tipo → ↑ **id**	{↑}	
tipo → **array** [*semplice*] **of** *tipo*	{**array**}	
semplice → **integer**	{**integer**}	
semplice → **char**	{**char**}	
semplice → **num . . num**	{**num**}	

e quindi la tabella di parsing predittivo (ignorando le colonne vuote):

variabile	**integer**	**char**	**num**	↑	**array**
tipo	*tipo* → *semplice*	*tipo* → *semplice*	*tipo* → *semplice*	*tipo* → ↑ **id**	*tipo* → **array** [*semplice*] **of** *tipo*
semplice	*semplice* → **integer**	*semplice* → **char**	*semplice* → **num . . num**		

Quando applichiamo la procedura alla stringa **array [num . . num] of integer** otteniamo esattamente l'albero di derivazione che abbiamo costruito in precedenza.
L'implementazione che segue è chiamata *analisi per discesa ricorsiva* (recursive-descent parsing) e consiste di un insieme di procedure, una per ogni simbolo nonterminale. La procedura associata al generico simbolo nonterminale *A* è la seguente.

(*selezione*) Applicare una delle produzioni con testa *A* facendo uso della tabella di parsing predittivo. Sia α il corpo della produzione scelta.

(*sviluppo*) Se α è la stringa vuota, allora far avanzare la sonda.

Altrimenti, sia $\alpha = X_1 \ldots X_n$.
Per $i = 1, \ldots, n$
se X_i è un simbolo nonterminale, allora richiamare la procedura associata ad X_i;
altrimenti,
se X_i è identico al simbolo corrente, far avanzare sia la sonda che il cursore;
altrimenti, segnalare 'Errore".

8.1.1 *Circolo vizioso*

L'analisi per discesa ricorsiva potrebbe non essere un processo algoritmico, nel senso che potrebbe non aver mai fine. Questo accade quando si finisce per entrare in un circolo vizioso per la presenza delle cosiddette produzioni "ricorsive a sinistra". Una produzione $A \rightarrow \alpha$ è *ricorsiva a sinistra* (per brevità, *ricorsiva*) se esiste una stringa derivabile da α che inizi con il simbolo A. Un esempio di produzione ricorsiva è quella che chiamiamo *produzione ricorsiva semplice*, cioè una produzione del tipo $A \rightarrow A\alpha$, come nella produzione `E` $\rightarrow$ `E+T`. È ovvio che, se durante la costruzione dell'albero di derivazione il nodo di controllo è etichettato da A, allora la produzione $A \rightarrow A\alpha$ viene utilizzata ad infinitum (circolo vizioso) senza che mai il cursore abbia modo di avanzare. D'altra parte, se $A \rightarrow A\alpha$ fosse l'unica produzione con testa A, allora la produzione $A \rightarrow A\alpha$ sarebbe inutile perché A sarebbe una variabile improduttiva di prima specie. Supponiamo allora di aver già eliminato tutte le produzioni inutili, cosicché, se esiste la produzione $A \rightarrow A\alpha$, allora deve esistere almeno una produzione $A \rightarrow \beta$ tale che β non inizi per A. Ora, si osservi che se $a \in \mathcal{I}(\beta)$ allora abbiamo pure che $a \in \mathcal{I}(A)$ e, quindi, $a \in \mathcal{I}(A\alpha)$, cosicché nella cella (A, a) della tabella di parsing predittivo compaiono sia la produzione $A \rightarrow A\alpha$ che la produzione $A \rightarrow \beta$. Ad esempio, consideriamo ancora le produzioni

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

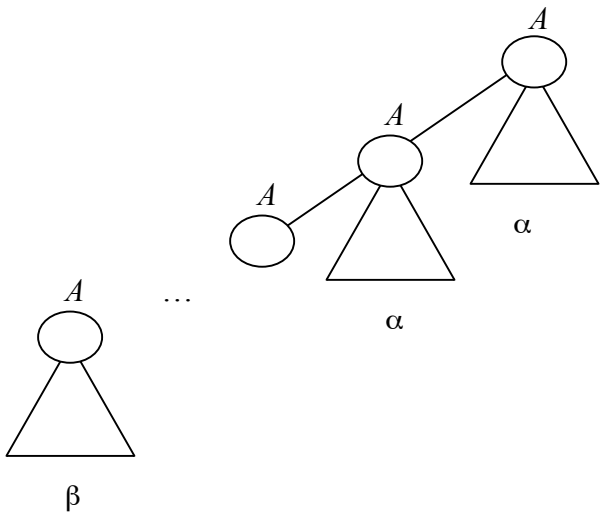
Sulla scorta dei seguenti valori delle funzioni $\mathcal{I}$ e $\mathcal{J}$:

$\mathcal{I}$(`E`) = $\mathcal{I}$(`T`) = $\mathcal{I}$(`F`) = { `(`, **id**} $\mathcal{J}$(`E`) = $\mathcal{J}$(`T`) = $\mathcal{J}$(`F`) = {`$`, `+`, `*`, `)`}

otteniamo la tabella di parsing predittivo:

	id	+	*	(	)	$
E	E → E+T | T			E+T | T		
T	T → T*F | F			T → T*F | F		
F	F → **id**			F → (E)		

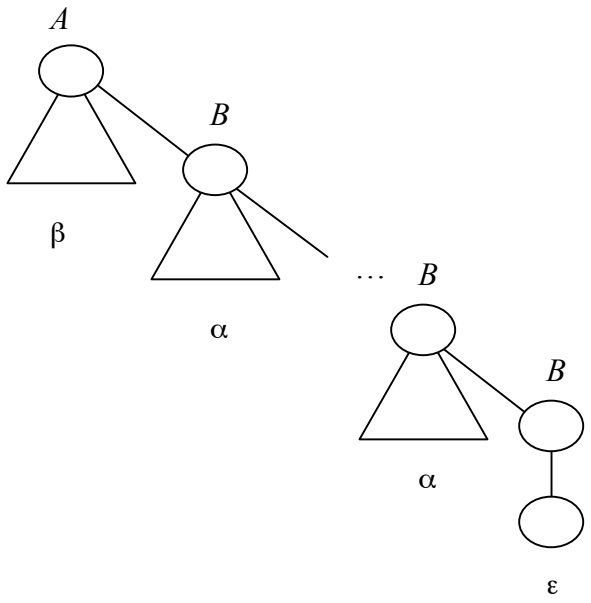
Fortunatamente, possiamo eliminare la ricorsione semplice nella produzione $A \rightarrow A\alpha$ grazie alla produzione $A \rightarrow \beta$. L'applicazione ripetuta della produzione $A \rightarrow A\alpha$ e poi della produzione $A \rightarrow \beta$ genera un sottoalbero della forma



Lo stesso risultato si ottiene con le tre produzioni

$A \rightarrow \beta\, B$
$B \rightarrow \alpha\, B$
$B \rightarrow \varepsilon$

dove B è un nuovo simbolo nonterminale, introdotto ad hoc.



Così le produzioni

E → E+T | T
T → T*F | F
F → (E) | **id**

sono equivalenti alle seguenti

E → TA
A → +TA | ε
T → FB
B → *FB | ε
F → (E) | **id**

Quando applichiamo gli Algoritmi 8.1, 8.2 e 8.3 otteniamo

$\mathcal{I}$(F) = $\mathcal{I}$(T) = $\mathcal{I}$(E) = {(, **id**} $\mathcal{I}$(A) = {+, ε} $\mathcal{I}$(B) = {*, ε}

$\mathcal{J}$(E) = $\mathcal{J}$(A) = {), $} $\mathcal{J}$(T) = $\mathcal{J}$(B) = {+,), $} $\mathcal{J}$(F) = {+, *,), $}

e finalmente la tabella di parsing predittivo:

	id	+	*	(	)	$
E	E → TA			E → TA		
A		A → +TA			A → ε	A → ε
T	T → FB			T → FB		
B		B → ε	B → *FB		B → ε	B → ε

F	F → **id**			F → (E)		

Nel caso più generale di una o più produzioni ricorsive semplici con testa A

$$A \rightarrow A\ \alpha_1 \mid \ldots \mid A\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

in cui nessuna stringa β_j ($1 \leq j \leq m$) inizia per A, l'equivalente insieme di produzioni è

$$A \rightarrow \beta_1\ B \mid \ldots \mid \beta_m\ B$$
$$B \rightarrow \alpha_1\ B \mid \ldots \mid \alpha_n\ B \mid \varepsilon$$

Passiamo ora al caso generale di produzioni ricorsive. Supporremo di aver già eliminato le produzioni nulle nonché le produzioni unitarie che rendono un simbolo nonterminale derivabile da se stesso. A questo punto si osservi che la grammatica che se ne ottiene non contiene produzioni ricorsive se, scelto arbitrariamente un ordinamento (A_1, …, A_n) dei simboli nonterminali, sussiste la seguente proprietà:

$\forall\ i, l$ se $A_i \rightarrow A_l\ \gamma$ è una produzione allora $l > i$.

Per ottenere una siffatta grammatica applicheremo l'algoritmo che segue.

Algoritmo 8.4

1. Scegliere un ordinamento dei simboli nonterminali, sia esso (A_1, …, A_n).

2. Per $i = 1, \ldots, n$

2.1. se $i > 1$ allora

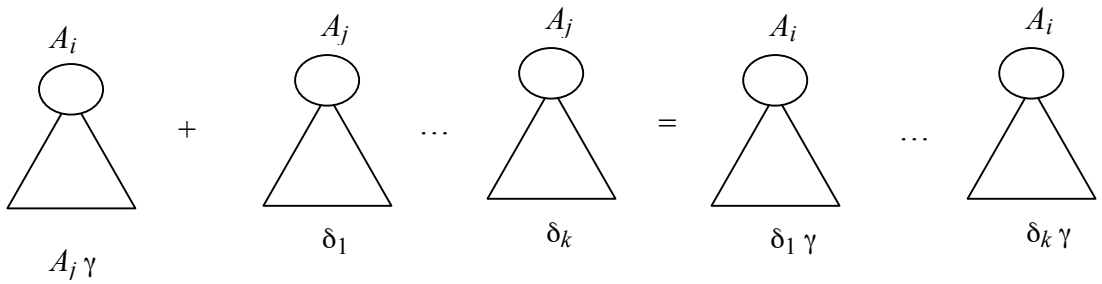
per $j = 1, \ldots, i-1$

sostituire ogni produzione della forma $A_i \rightarrow A_j\ \gamma$ con le produzioni

$$A_i \rightarrow \delta_1\ \gamma \mid \ldots \mid \delta_k\ \gamma$$

dove $\delta_1, \ldots, \delta_k$ sono i corpi delle produzioni con testa A_j ($A_j \rightarrow \delta_1 \mid \ldots \mid \delta_k$);

2.2. eliminare le produzioni ricorsive semplici con testa A_i.



La correttezza dell'Algoritmo 8.4 può essere provata per induzione su i.

BASE. Per $i = 1$, al passo 2.2 vengono eliminate tutte le produzioni ricorsive semplici con testa A_1. Pertanto, se $A_1 \rightarrow A_l\,\gamma$ è una produzione allora $l > 1$.

PASSO INDUTTIVO. Consideriamo il caso $i > 1$. Per ipotesi, abbiamo che per ogni $j < i$ è vero che se $A_j \rightarrow A_l\,\gamma$ è una produzione allora $l > j$. È sufficiente dimostrare che, dopo aver eseguito il passo 2.1, se $A_i \rightarrow A_l\,\gamma$ è una produzione allora $l \geq i$; infatti, le (eventuali) produzioni ricorsive semplici del tipo $A_i \rightarrow A_i\,\gamma$ verranno tutte eliminate al passo 2.2.
Iniziamo da $j = 1$. Se esiste una produzione del tipo $A_i \rightarrow A_1\,\gamma$ assieme alle produzioni $A_1 \rightarrow \delta_1 \mid \ldots \mid \delta_k$, allora la produzione $A_i \rightarrow A_1\,\gamma$ viene sostituita con le produzioni $A_i \rightarrow \delta_1\,\gamma \mid \ldots \mid \delta_k\,\gamma$. Si osservi che, per ogni h, $1 \leq h \leq k$, se il simbolo iniziale di δ_h è il simbolo nonterminale A_l, allora $l > 1$ per l'ipotesi induttiva. Così, a questo punto possono solo aversi produzioni del tipo $A_i \rightarrow A_l\,\gamma$ con $l > 1$.
Passiamo a $j = 2$. Se esiste una produzione del tipo $A_i \rightarrow A_2\,\gamma$ assieme alle produzioni $A_2 \rightarrow \delta_1 \mid \ldots \mid \delta_k$, allora la produzione $A_i \rightarrow A_2\,\gamma$ viene sostituita con le produzioni $A_i \rightarrow \delta_1\,\gamma \mid \ldots \mid \delta_k\,\gamma$. Di nuovo, per ogni h, $1 \leq h \leq k$, se il simbolo iniziale di δ_h è il simbolo nonterminale A_l, allora $l > 2$ per l'ipotesi induttiva. Così, a questo punto possono solo aversi produzioni del tipo $A_i \rightarrow A_l\,\gamma$ con $l > 2$.
Ripetendo lo stesso ragionamento per $j = 3, \ldots, i–1$, possiamo concludere che, dopo aver eseguito il passo 2.1, se troviamo una produzione $A_i \rightarrow A_l\,\gamma$, allora $l > i–1$, cioè $l \geq i$, che è quanto volevamo dimostrare.

Esempio 8.1 Consideriamo la grammatica

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

Dopo aver eliminato la produzione nulla otteniamo

$$S \rightarrow Aa \mid a \mid b$$
$$A \rightarrow Ac \mid Sd \mid c$$

Le produzioni $S \rightarrow Aa$ e $A \rightarrow Ac$ sono entrambe ricorsive e la seconda è una produzione ricorsiva semplice. Applichiamo l'Algoritmo 8.4 con l'ordinamento (S, A). Al passo 2 per $i = 1$ non cambia nulla perché non esistono produzioni ricorsive semplici con testa S. Ma, per $i = 2$ andremo a sostituire la produzione $A \rightarrow Sd$ con le produzioni

$$A \rightarrow Aad \mid ad \mid bd$$

così che la grammatica diventa

$$S \rightarrow Aa \mid a \mid b$$
$$A \rightarrow Ac \mid Aad \mid ad \mid bd \mid c$$

Infine, andremo ad eliminare le due produzioni ricorsive semplici, così che la grammatica diventa

$$S \rightarrow Aa \mid a \mid b$$
$$A \rightarrow adB \mid bdB \mid cB$$
$$B \rightarrow cB \mid adB \mid \varepsilon$$

Quando applichiamo gli Algoritmi 8.1, 8.2 e 8.3, otteniamo la tabella $\mathcal{I}$-$\mathcal{J}$

$A \rightarrow \alpha$	$\mathcal{I}(\alpha)$	$\mathcal{J}(A)$
$S \rightarrow Aa$	$\{a, b, c\}$	
$S \rightarrow a$	$\{a\}$	
$S \rightarrow b$	$\{b\}$	
$A \rightarrow adB$	$\{a\}$	
$A \rightarrow bdB$	$\{b\}$	
$A \rightarrow cB$	$\{c\}$	
$B \rightarrow cB$	$\{c\}$	
$B \rightarrow adB$	$\{a\}$	
$B \rightarrow \varepsilon$	$\{\varepsilon\}$	$\{a\}$

e, quindi, la tabella di parsing predittivo (ignorando le colonne vuote):

	a	*b*	*c*
S	$S \to Aa$ $S \to a$	$S \to Aa$ $S \to b$	$S \to Aa$
A	$A \to adB$	$A \to bdB$	$A \to cB$
B	$B \to adB$ $B \to \varepsilon$		$B \to cB$

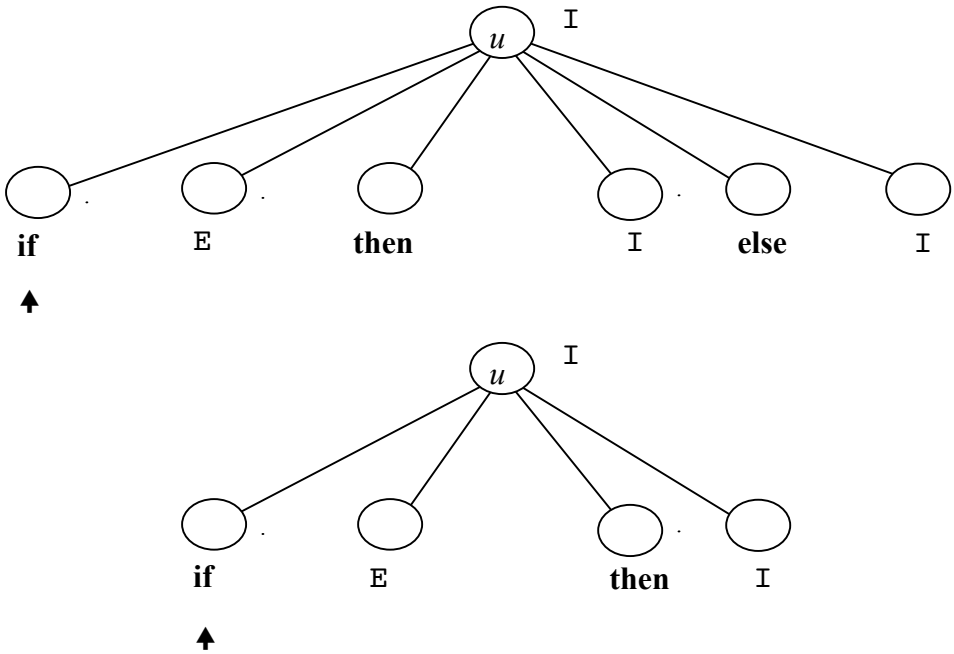
8.1.2 *Incertezza nella selezione*

L'analisi per discesa ricorsiva può riservare sorprese inaspettate se nella tabella di parsing predittivo una cella contiene due o più produzioni. Supponiamo che, data una stringa, ad un certo istante del processo di analisi il simbolo corrente *a* occupi la posizione *i*-esima del buffer, che il nodo di controllo *u* sia etichettato da un simbolo nonterminale *A* e che $a \in \mathcal{I}(\alpha) \cap \mathcal{I}(\beta)$, cosicché la cella (A, a) della tabella di parsing predittivo contiene sia la produzione $A \to \alpha$ che la produzione $A \to \beta$. Ammettiamo che venga selezionata la produzione $A \to \alpha$ (e la cosa non cambia se ad essere selezionata fosse la produzione $A \to \beta$). Allora può succedere che si finisca in un vicolo cieco e questo avviene se, dopo uno o più passi, il nodo di controllo viene ad essere un vertice etichettato da un simbolo terminale diverso dal simbolo corrente. In tal caso, si può (*a posteriori*) dire che la scelta della produzione $A \to \alpha$ era sbagliata cosicché, prima di provare con la produzione $A \to \beta$, va ripristinato lo "status quo ante" riportando indietro il cursore (△) nella posizione *i*-esima della stringa d'ingresso e facendo risalire la sonda (🡅) al vertice *u*; dopodiché, andremo a sviluppare il vertice *u* utilizzando la produzione $A \to \beta$. In altri termini, siamo stati costretti a "tornare sui nostri passi" (backtracking) per imboccare un'altra strada dopo aver arretrato sia il cursore che la sonda. Per ridurre la lunghezza del cammino a ritroso, si fa uso della tecnica chiamata "raccoglimento a sinistra del fattor comune" ed illustrata dall'esempio che segue.

Consideriamo la grammatica (ambigua) contenente le due produzioni

`I` → **if** `E` **then** `I` **else** `I` | **if** `E` **then** `I`

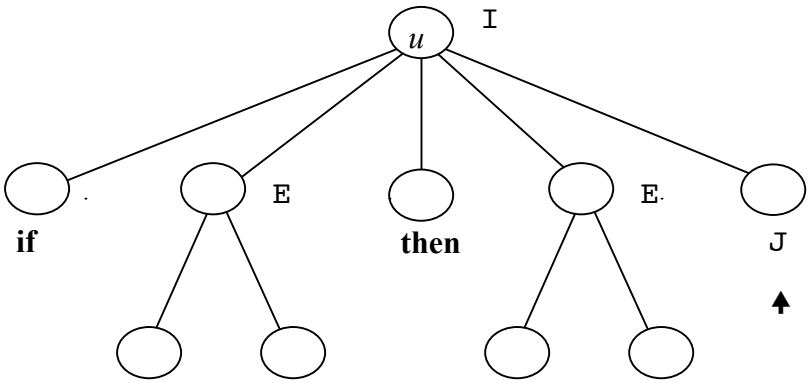
Supponiamo che nella costruzione di un albero di derivazione il nodo di controllo *u* sia etichettato da `I` e che il simbolo corrente sia **if**. Ovviamente, qui abbiamo due alternative



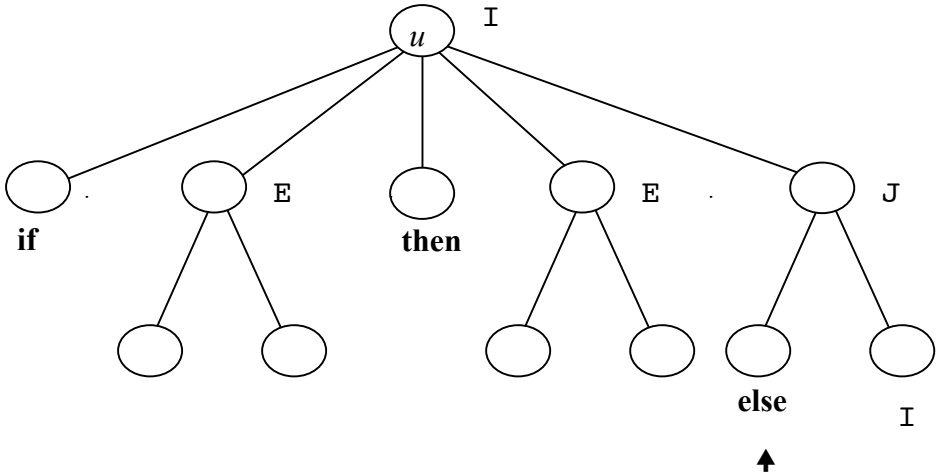
Ora, se utilizziamo la produzione `I` → **if** `E` **then** `I` **else** `I`, allora dopo aver raggiunto il vertice etichettato da **else** potremmo trovarci nella situazione che il simbolo corrente sia diverso da **else** cosa che ci obbligherebbe a far risalire la sonda al vertice *u* ed a far arretrare il cursore alla posizione del simbolo **if**. Questo è il caso della stringa di ingresso **if** `E` **then** `I ;`. D'altra parte, se utilizziamo la produzione `I` → **if** `E` **then** `I`, allora dopo aver raggiunto il secondo vertice etichettato da `I`, potremmo trovarci nella situazione che il simbolo corrente sia **else** cosa che ci obbligherebbe di nuovo a risalire la sonda al vertice *u* e di far arretrare il cursore alla posizione del simbolo **if**. Per limitare i danni, conviene trasformare le due produzioni raccogliendo il fattor comune **if** `E` **then** `I`:

`I` → **if** `E` **then** `I J`
`J` → **else** `I` | ε

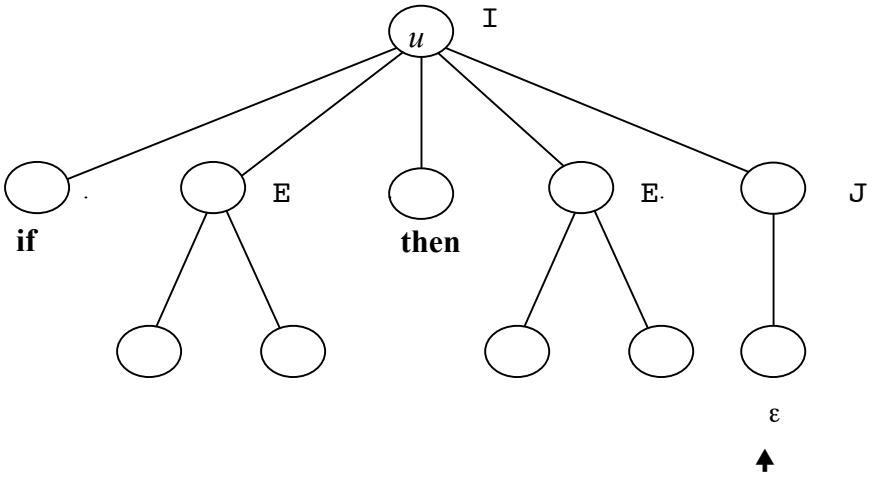
Così facendo si procrastina la scelta tra le due alternative al momento in cui la sonda raggiunge il vertice etichettato da `J`.



A questo punto, se il simbolo corrente è **else**, allora correttamente viene usata la produzione J → **else** I.

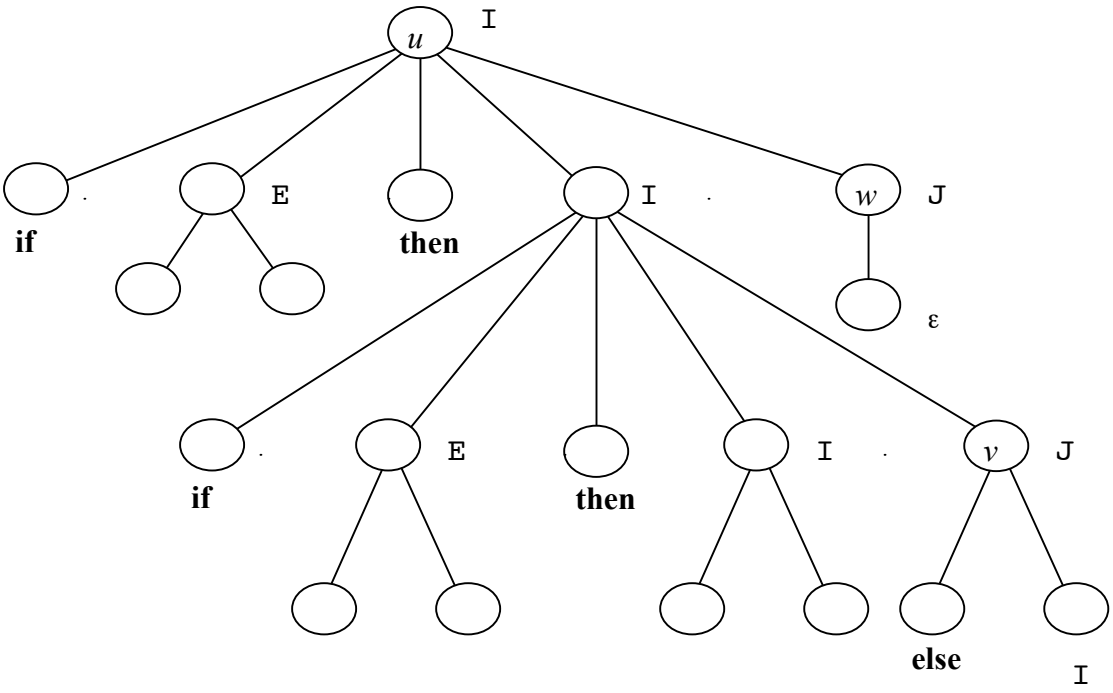


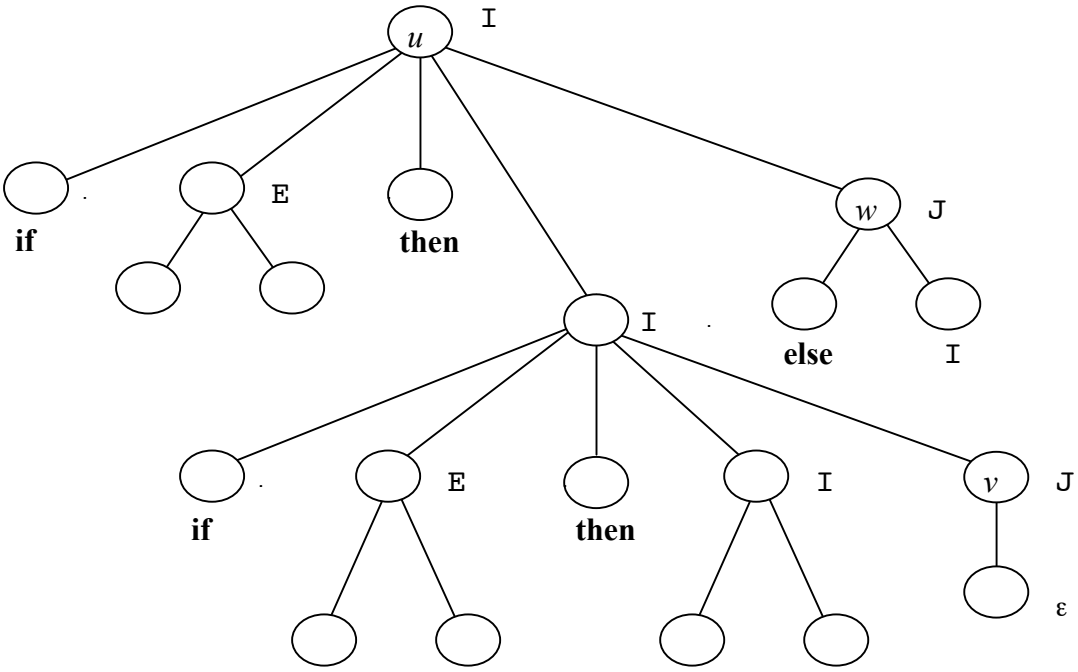
Altrimenti, per esempio se il simbolo corrente è il punto di interpunzione **;**, allora viene usata la produzione J → ε



e la costruzione continua, senza far avanzare il cursore, dal vertice successivo al vertice *u*.

Si osservi infine che questa trasformazione non fa che rimandare il punto di ambiguità come dimostra la stringa **if** `E` **then if** `E` **then** `I` **else** `I` per la quale abbiamo due distinti alberi di derivazione corrispondenti alle due interpretazioni **if** `E` **then** (**if** `E` **then** `I` **else** `I`) e **if** `E` **then** (**if** `E` **then** `I`) **else** `I`:





Una maniera per risolvere una tale ambiguità consiste nell'associare **else** all'ultima delle occorrenze di **then** nella stringa che lo precede (v. primo albero di derivazione).

8.1.3 *Grammatiche LL*(1)

Nel caso in cui la tabella di parsing predittivo contiene al più una stringa in ogni cella, l'analisi per discesa ricorsiva è un processo deterministico che prende il nome di *analisi predittiva* e la grammatica (come quella generativa dei tipi) è detta essere di *tipo LL*(1). Formalmente, una grammatica $\boldsymbol{G} = (N, T, P, S)$ è di tipo $LL(1)$ se è priva di produzioni ricorsive a sinistra e, per ogni coppia di produzioni distinte

$$A \rightarrow \alpha \qquad A \rightarrow \beta,$$

sono soddisfatte tutte e tre le condizioni:

— $\mathcal{I}(\alpha) \cap \mathcal{I}(\beta) = \varnothing$,

— se $\varepsilon \in \mathcal{I}(\beta)$ allora $\mathcal{I}(\alpha) \cap \mathcal{J}(A) = \varnothing$

— se $\varepsilon \in \mathcal{I}(\alpha)$ allora $\mathcal{I}(\beta) \cap \mathcal{J}(A) = \varnothing$.

Anche se la classe delle grammatiche $LL(1)$ è abbastanza ampia da includere le grammatiche di molti linguaggi di programmazione, vi sono grammatiche acontestuali che non appartengono alla classe delle grammatiche $LL(1)$. Esempio ne sono le grammatiche ambigue e quelle che contengono produzioni ricorsive a sinistra di cui abbiamo già parlato.

Vediamo infine un'implementazione dell'analisi predittiva che produce una derivazione sinistrorsa di una frase x. Il metodo fa uso

di un buffer di input,
della tabella di parsing predittivo e
di un dispositivo di memoria a pila.

Il contenuto del buffer ad un generico istante è una stringa y`$`, dove y è il suffisso di x formato dai simboli che rimangono da leggere. Così, y è la stringa vuota se e solo se il simbolo corrente (il simbolo indicato dal cursore) è il carattere `$`.

Il contenuto della pila è sempre una stringa di simboli terminali e non, seguita dal carattere `$` che è l'elemento in fondo della pila. Inizialmente, il contenuto della pila è la stringa S`$` dove S è il simbolo iniziale della grammatica e il contenuto del buffer è x`$`. Al generico istante, sia X l'elemento apicale della pila e sia a il simbolo corrente.

Fintantoché $X \neq$ `$` ripetere

se $X = a$ allora

eseguire l'azione "accetta a" facendo avanzare il cursore;
eliminare l'elemento apicale della pila;

altrimenti,

se X è un simbolo terminale diverso da a oppure è un simbolo nonterminale e la cella (X, a) della tabella di parsing predittivo è vuota, allora segnalare 'errore';

altrimenti

sia $X \rightarrow \alpha$ è la produzione contenuta nella cella (X, a);
eseguire l'azione "applica $X \rightarrow \alpha$" che consiste nell'eliminazione dell'elemento apicale X della pila e, se $\alpha \neq \varepsilon$, nell'inserimento nella pila dei singoli simboli che compongono α in ordine inverso rispetto alla loro posizione in α (cioè, se $\alpha = X_1 \ldots X_{m-1} X_m$ allora X_m viene aggiunto alla pila per primo e X_1 per ultimo);

porre X uguale all'elemento apicale della pila.

Per illustrare la procedura, riprendiamo le produzioni

`E` → `TA`
`A` → `+TA` | ε
`T` → `FB`
`B` → `*FB` | ε
`F` → `(E)` | **id**

e la tabella di parsing predittivo:

	id	+	*	(	)	$
E	E → TA			E → TA		
A		A → +TA			A → ε	A → ε
T	T → FB			T → FB		
B		B → ε	B → *FB		B → ε	B → ε
F	F → **id**			F → (E)		

viste al paragrafo 8.1.1, ed analizziamo la stringa **id+id*id**. La sequenza dei passi è mostrata in Figura.

azione	*sequenza dei simboli accettati*	*pila*	*buffer*
		E$	**id+id*id**$
applica E → TA		TA$	**id+id*id**$
applica T → FB		FBA$	**id+id*id**$
applica F → **id**		**id**BA$	**id+id*id**$
accetta **id**	**id**	BA$	**+id*id**$
applica B → ε	**id**	A$	**+id*id**$
applica A → +TA	**id**	+TA$	**+id*id**$
accetta +	**id+**	TA$	**id*id**$
applica T → FB	**id+**	FBA$	**id*id**$
applica F → **id**	**id+**	**id**BA$	**id*id**$
accetta **id**	**id+id**	BA$	***id**$
applica T → *FB	**id+id**	*FBA$	***id**$
accetta *	**id+id***	FBA$	**id**$
applica F → **id**	**id+id***	**id**BA$	**id**$
accetta **id**	**id+id*id**	BA$	$
applica B → ε	**id+id*id**	A$	$
applica A → ε	**id+id*id**	$	$

Le mosse del parsing predittivo corrispondono alla derivazione sinistrorsa

(E, TA, FBA, **id** BA, **id** A, **id+** TA, …)

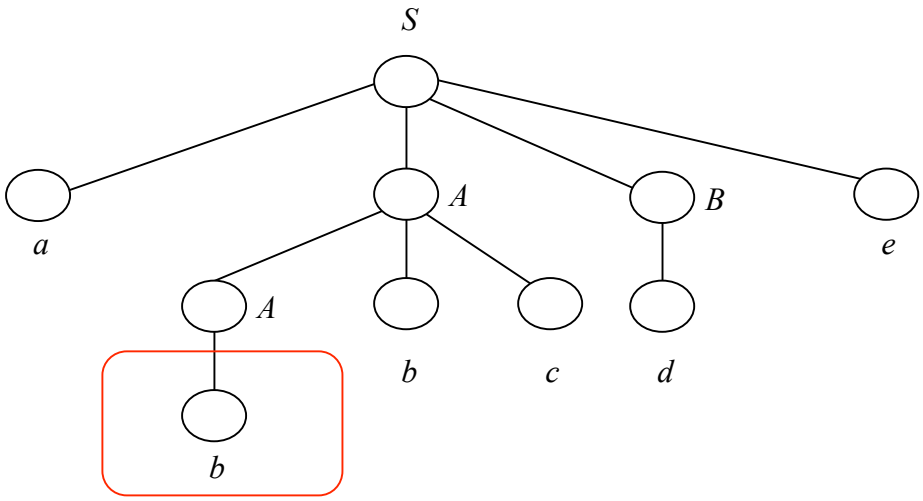
e le formule in questa derivazione si ottengono concatenando il prefisso di *x* formato dai simboli già accettati (seconda colonna) con la stringa contenuta nella pila (terza colonna) a meno di $.

8.2 Approccio induttivo

I metodi induttivi mirano a costruire una derivazione destrorsa della stringa in esame. Per dare un'idea della tecnica che viene impiegata, consideriamo la grammatica acontestuale

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

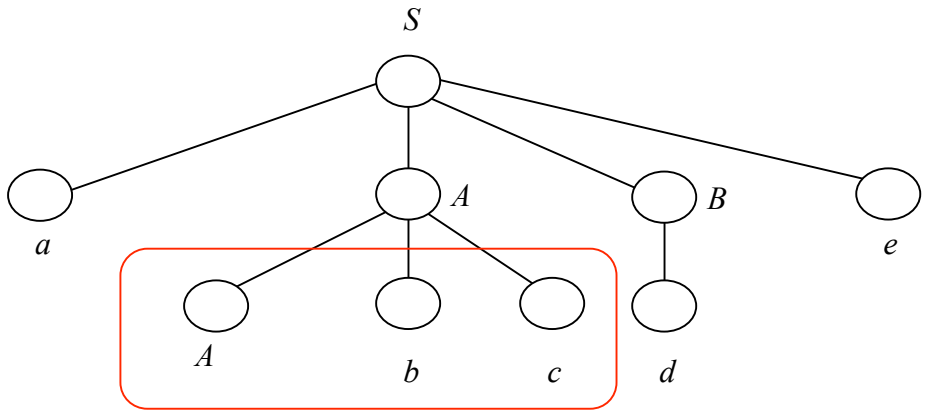
La stringa *abbcde* è una frase della grammatica ed un albero di derivazione T è mostrato in figura.



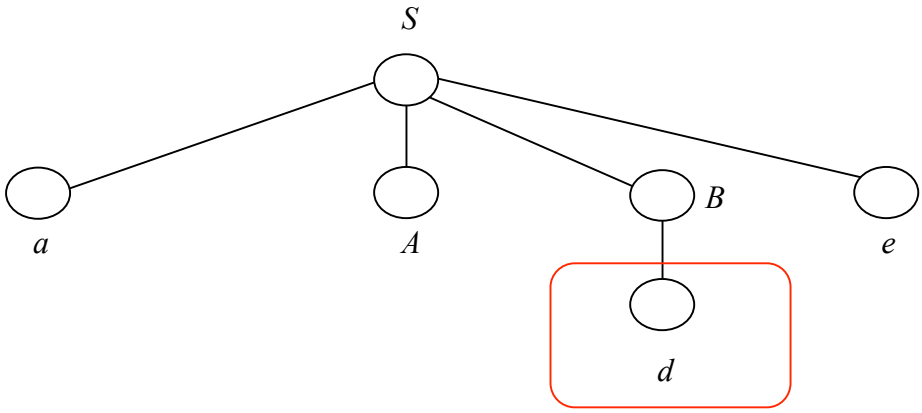
La derivazione destrorsa associata a T è (*S*, *aABe*, *aAde*, *aAbcde*, *abbcde*). Consideriamo la sua sequenza inversa

(*abbcde*, *aAbcde*, *aAde*, *aABe*, *S*).

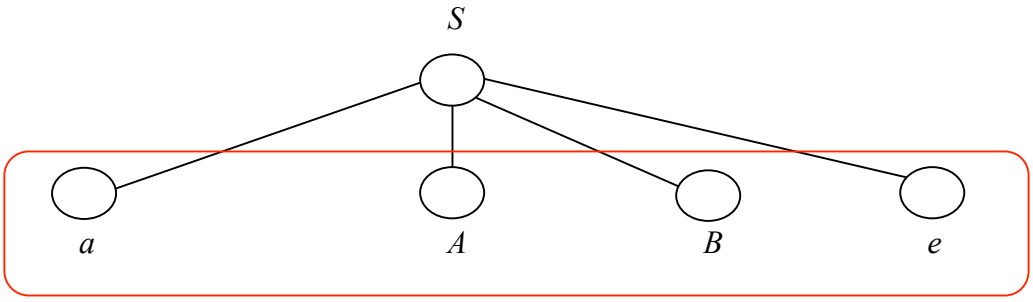
che chiamiamo una *coderivazione* (*destrorsa*) della stringa *abbcde*. Ovviamente, la prima stringa (*abbcde*) della coderivazione è la concatenazione delle etichette delle foglie dell'albero $T^{(1)} = T$. Le altre stringhe della coderivazione possono ancora interpretarsi come la concatenazione delle etichette delle foglie di *alberi ridotti*, che si ottengono per "potatura" di $T^{(1)}$. Ad esempio, la seconda stringa (*aAbcde*) è la concatenazione delle etichette delle foglie dell'albero $T^{(2)}$ che si ottiene dopo aver potato la seconda foglia di $T^{(1)}$.



La terza (*aAde*) è la concatenazione delle etichette delle foglie dell'albero $T^{(3)}$ che si ottiene dopo aver potato la seconda, la terza e la quarta foglia di $T^{(2)}$.



La quarta (*aABe*) è la concatenazione delle etichette delle foglie dell'albero $T^{(4)}$ che si ottiene dopo aver potato la terza foglia di $T^{(3)}$.



La quinta (*S*) è l'etichetta dell'unico vertice dell'albero $T^{(5)}$ che si ottiene dopo aver potato le foglie di $T^{(4)}$.

Vedremo come, partendo soltanto dalla stringa *abbcde*, possiamo costruire una sua coderivazione e, quindi, una sua derivazione destrorsa.

In generale, data una frase x di una grammatica acontestuale ed una coderivazione (σ_k, σ_{k-1}, …, σ_0) di x, se $\sigma_h = \lambda\alpha y$ e $\sigma_{h-1} = \lambda A y$ chiamiamo la stringa α e la produzione $A \rightarrow \alpha$ rispettivamente *oggetto della riduzione* e *produzione riducente*.

Vediamo un altro esempio questa volta con una grammatica ambigua:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Per la stringa

id + id * id

abbiamo due coderivazioni. Una è

(**id+id*id**, *E*+**id*id**, *E*+*E***id**, *E*+*E***E*, *E*+*E*, *E*)

ed il corrispondente processo di riduzione si sviluppa come segue

formula	*oggetto della riduzione*	*produzione riducente*
id+id*id	**id**	$E \rightarrow$ **id**
E+**id*id**	**id**	$E \rightarrow$ **id**
E+*E***id**	**id**	$E \rightarrow$ **id**
E+*E***E*	*E***E*	$E \rightarrow E * E$
E+*E*	*E*+*E*	$E \rightarrow E + E$
E		

L'altra è

(**id+id*id**, *E*+**id*id**, *E*+*E***id**, *E***id**, *E***E*, *E*)

ed il corrispondente processo di riduzione si sviluppa come segue

formula	*oggetto della riduzione*	*produzione riducente*
id$_1$**+id*id**	**id**	$E \rightarrow$ **id**
E+**id*id**	**id**	$E \rightarrow$ **id**
E+*E***id**	*E*+*E*	$E \rightarrow E + E$
*E***id**	**id**	$E \rightarrow$ **id**
*E***E*	*E***E*	$E \rightarrow E * E$
E		

Ora, data una stringa x, che potrebbe anche non essere una frase della grammatica, vediamo come proveremo a costruire una coderivazione $(\sigma_k, \sigma_{k-1}, \ldots, \sigma_0)$ di x. Ovviamente, la procedura terminerà con successo se e solo se x è una frase della grammatica. Naturalmente, la prima stringa della sequenza è $\sigma_k = x$. Assumiamo per il momento che x sia una frase della grammatica e supponiamo di aver già costruito la sequenza $(\sigma_k, \sigma_{k-1}, \ldots, \sigma_h)$, $1 \leq h \leq k$, che sia la parte iniziale di una coderivazione di x. Per poter estendere la sequenza, dobbiamo trovare una sottostringa α di σ_h tale che $\sigma_h = \lambda\alpha y$ ed esista una produzione $A \rightarrow \alpha$ della grammatica; allora, andremo ad aggiungere in coda alla sequenza $(\sigma_k, \sigma_{k-1}, \ldots, \sigma_h)$ la stringa $\lambda A y$. (Si osservi che, siccome la derivazione di x è destrorsa, A deve essere il simbolo nonterminale più a

destra di λAy e, quindi, y deve essere una stringa di simboli terminali.) Estendere la sequenza $(\sigma_k, \sigma_{k-1}, \ldots, \sigma_h)$ in maniera tale che $(\sigma_k, \sigma_{k-1}, \ldots, \sigma_h, \sigma_{h-1})$ sia ancora una porzione di una coderivazione non è affatto cosa semplice perché la stringa σ_{h-1} che si ottiene potrebbe non essere una formula e, se lo è, potrebbe non ammettere una derivazione destrorsa dal simbolo iniziale della grammatica sicché il completamento della sequenza $(\sigma_k, \sigma_{k-1}, \ldots, \sigma_h, \sigma_{h-1})$ è destinato prima o poi a fallire. Tornando al penultimo esempio, notiamo che, al primo passo della procedura, con $\sigma_k = abbcde$ abbiamo tre alternative per σ_{k-1}:

$\sigma_{k-1} = aAbcde$	l'oggetto della riduzione è la prima occorrenza del simbolo b e la produzione riducente è $A \to b$
$\sigma_{k-1} = abAcde$	l'oggetto della riduzione è la seconda occorrenza del simbolo b e la produzione riducente è $A \to b$
$\sigma_{k-1} = abbcBe$	l'oggetto della riduzione è l'unica occorrenza del simbolo d e la produzione riducente è $B \to d$

ma solo a posteriori ci accorgeremo che

la stringa *aAbcde* è effettivamente una formula destrorsa (v. sopra),

la stringa *aaAcde* non è una formula, e

la stringa *abbcBe* è sì una formula perché (*S*, *aABe*, *aAbcBe*, *abbcBe*) ne è una derivazione, ma non è una derivazione destrorsa.

È dunque importante saper distinguere tra le formule quelle che ammettono una derivazione destrorsa. Le chiameremo *formule destrorse*. Nel prossimo paragrafo, daremo un metodo per estendere correttamente una sottosequenza $(\sigma_k, \sigma_{k-1}, \ldots, \sigma_h)$, $h \leq k$, in maniera tale da costruire una coderivazione di x sempreché x sia una frase della grammatica.

8.2.1 *Procedura di riduzione*

Vediamo ora un'implementazione del metodo induttivo, che prende il nome di *procedura di riduzione* (shift-reduce parsing) e risulta in genere più potente dell'analisi per discesa ricorsiva e si applica alla maggior parte delle grammatiche acontestuali generative di linguaggi di programmazione. Innanzitutto, data una grammatica acontestuale $\boldsymbol{G} = (N, T, P, S)$, vengono costruiti

— un automa finito quasi-deterministico $\boldsymbol{D}$ sull'alfabeto $N \cup T$,

— ed una *tabella delle azioni* che associa alle coppie $(q, e) \in Q \times (T \cup \{\$\})$ una delle quattro azioni:

T	(trasferimento nella pila),
R($A\to\alpha$)	(riduzione mediante la produzione $A\to\alpha$ della grammatica),
A	(accettazione),
E	(errore).

Inoltre, la procedura di riduzione fa anche uso

di un dispositivo di memoria a pila, e
di un buffer di input.

Il contenuto della pila è sempre una sequenza del tipo

$$(q_0, X_1, q_1, X_2, \ldots, q_{m-1}, X_m, q_m),$$

in cui ogni q_i è uno stato dell'automa ***D*** (di cui q_0 è lo stato iniziale) e ogni X_i è un simbolo (terminale o non) della grammatica ***G***.

Per una assegnata stringa x su T (la stringa in esame), rappresentiamo il contenuto del buffer ad un generico istante come y\$, dove y è il suffisso di x formato dai simboli che rimangono da leggere. Così, y è la stringa vuota se e solo se il simbolo corrente (il simbolo indicato dal cursore) è il carattere \$.

Inizialmente, il contenuto della pila è (q_0) ed il contenuto del buffer è x\$. Al generico istante del processo di riduzione, il contenuto della pila (q_0, X_1, q_1, X_2, …, q_{m-1}, X_m, q_m) ed il contenuto del buffer y\$ sono tali che la stringa $X_1X_2\ldots X_my$ sia una formula destrorsa di ***G***. Chiameremo $X_1X_2\ldots X_m$ la *stringa contenuta nella pila*. A questo punto, la scelta dell'azione da eseguire è dettata dalla coppia (q_m, e) dove q_m è lo stato in cui si trova l'automa ***D*** ed e ($\in T\cup\{\$\}$) è il simbolo corrente nel buffer. Discutiamo ora i quattro casi che si possono presentare a seconda dell'azione associata alla coppia (q_m, e):

Caso 1: l'azione associata alla coppia (q_m, e) è T. In tal caso

— $e \neq \$$
— l'automa ***D*** effettua una transizione con stato finale $q = \delta(q_m, e)$
— il contenuto della pila diventa $(q_0, X_1, q_1, X_2, \ldots, q_{m-1}, X_m, q_m, e, q)$
— il cursore del buffer viene fatto avanzare di una posizione.

Caso 2: l'azione associata alla coppia (q_m, e) è R($A\to\alpha$). In tal caso

— α è un suffisso della stringa contenuta nella pila, cioè $\alpha = \varepsilon$ oppure $\alpha = X_{m-|\alpha|+1}X_{m-|\alpha|+2}\ldots X_m$
— l'automa ***D*** viene prima riportato nello stato $q_{m-|\alpha|}$, a partire dal quale effettua poi una transizione con stato finale $q = \delta(q_{m-|\alpha|}, A)$

— il contenuto della pila diventa $(q_0, X_1, q_1, X_2, \ldots, q_{m-|\alpha|}, A, q)$, cioè vengono eliminati i $2|\alpha|$ elementi apicali della pila $X_{m-|\alpha|+1}, q_{m-|\alpha|+1}, \ldots, X_m$ e, poi, vengono impilati A e q (in questo ordine)
— il cursore del buffer non viene fatto avanzare.

Caso 3: l'azione associata alla coppia (q_m, e) è A. In tal caso e = $ e la procedura termina con esito positivo, cioè la stringa x è riconosciuta come una frase della grammatica ***G***.

Caso 4: l'azione associata alla coppia (q_m, e) è E. In tal caso la procedura termina con esito negativo.

Ad esempio, per ottenere la coderivazione

$$(\mathbf{id}_1\mathbf{+id}_2\mathbf{*id}_3,\ E\mathbf{+id}_2\mathbf{*id}_3,\ E\mathbf{+}E\mathbf{*id}_3,\ E\mathbf{+}E\mathbf{*}E,\ E\mathbf{+}E,\ E)$$

della stringa $\mathbf{id}_1\mathbf{+id}_2\mathbf{*id}_3$, la procedura di riduzione effettua le seguenti azioni.

stringa contenuta nella pila	*contenuto del buffer*	*azione*
ε	$\mathbf{id}_1\mathbf{+id}_2\mathbf{*id}_3$$	T
$\mathbf{id}_1$	$\mathbf{+id}_2\mathbf{*id}_3$$	R(*E*→**id**)
E	$\mathbf{+id}_2\mathbf{*id}_3$$	T
E+	$\mathbf{id}_2\mathbf{*id}_3$$	T
E+$\mathbf{id}_2$	$\mathbf{*id}_3$$	R(*E*→**id**)
E+*E*	$\mathbf{*id}_3$$	T
E+*E**	$\mathbf{id}_3$$	T
E+*E**$\mathbf{id}_3$	$	R(*E*→**id**)
E+*E***E*	$	R(*E*→*E***E*)
E+*E*	$	R(*E*→*E*+*E*)
E	$	A

Si osservi che la coderivazione si ottiene riportando le stringhe che si ottengono concatenando la stringa contenuta nella pila con la stringa contenuta del buffer a meno del segno $, e ignorando stringhe consecutive uguali.

Vediamo ora con un esempio di grammatica acontestuale un po' meno semplice l'automa ***D*** e la tabella delle azioni. Sia ***G*** la grammatica con produzioni:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow \mathbf{id}$

La tabella delle transizioni dell'automa ***D*** è

stato	**id**	+	*	(	)	*E*	*T*	*F*
q_0	q_5			q_4		q_1	q_2	q_3
q_1		q_6						
q_2			q_7					
q_3								
q_4	q_5			q_4		q_8	q_2	q_3
q_5								
q_6	q_5			q_4			q_9	q_3
q_7	q_5			q_4				q_{10}
q_8		q_6			q_{11}			
q_9			q_7					
q_{10}								
q_{11}								

e la tabella delle azioni (dove, per comodità, le celle contenenti E sono lasciate vuote) è

stato	**id**	+	*	(	)	$
q_0	T			T		
q_1		T				A
q_2		R(2)	T		R(2)	R(2)
q_3		R(4)	R(4)		R(4)	R(4)
q_4	T			T		
q_5		R(6)	R(6)		R(6)	R(6)
q_6	T			T		
q_7	T			T		
q_8		T			T	
q_9		R(1)	T		R(1)	R(1)
q_{10}		R(3)	R(3)		R(3)	R(3)
q_{11}		R(5)	R(5)		R(5)	R(5)

Così, ad esempio, la procedura di riduzione della stringa **id*id+id** si sviluppa come segue.

pila	*buffer*	*azione*
q_0	**id*id+id**$	T
q_0**id**q_5	***id+id**$	R(6)
q_0Fq_3	***id+id**$	R(4)
q_0Tq_2	***id+id**$	T
q_0Tq_2***q_7	**id+id**$	T
q_0Tq_2***q_7**id**q_5	**+id**$	R(6)
q_0Tq_2***q_7Fq_{10}	**+id**$	R(3)
q_0Tq_2	**+id**$	R(2)
q_0Eq_1	**+id**$	T
q_0Eq_1**+**q_6	**id**$	T
q_0Eq_1**+**q_6**id**q_5	$	R(6)
q_0Eq_1**+**q_6Fq_3	$	R(4)
q_0Eq_1**+**q_6Tq_9	$	R(1)
q_0Eq_1	$	A

Nei prossimi paragrafi spiegheremo come viene costruito l'automa ***D*** e la tabella delle azioni.

8.2.2 *L'automa finito quasi-deterministico* ***D***

L'automa ***D*** è progettato per accettare tutte e solo le stringhe contenute nella pila durante l'applicazione della procedura di riduzione ad una qualsiasi stringa. Inoltre, ogni stato di ***D*** è uno stato di accettazione. Così, se ad un certo istante avremo che il contenuto della pila è (q_0, X_1, q_1, X_2, …, q_{m-1}, X_m, q_m), allora la stringa $X_1X_2…X_m$ contenuta nella pila è una stringa accettata da ***D*** e q_m è lo stato di ***D*** che si raggiunge dallo stato iniziale q_0 di ***D*** con la sequenza di transizioni etichettate nell'ordine con X_1, X_2, …, X_m. Inoltre, se la stringa in esame x è una frase della grammatica, allora la concatenazione di $X_1X_2…X_m$ con la parte residua di x contenuta nel buffer è sempre una formula destrorsa della grammatica che compare nella derivazione destrorsa di x.

Vogliamo ora definire il linguaggio accettato da ***D*** in maniera formale. A tale scopo, dobbiamo introdurre il concetto di "preformula" (viable prefix) della grammatica. Una stringa (di simboli terminali e non) è una *preformula* della grammatica se è il simbolo speciale della grammatica oppure è della forma

$\lambda\beta$

ed esistono

— una produzione $A \rightarrow \beta\gamma$ della grammatica e

— una formula destrorsa della grammatica del tipo $\lambda\beta\gamma y$.

In tal caso, anche la stringa λAy è una formula destrorsa della grammatica. Per evidenziare la posizione della stringa β nel corpo della produzione $A \rightarrow \beta\gamma$, diremo che la *fase* (item) $A \rightarrow \beta\bullet\gamma$ della produzione $A \rightarrow \beta\gamma$ è *compatibile* con la preformula $\lambda\beta$.

Si osservi che una produzione $A \rightarrow \alpha$ ha esattamente n+1 fasi dove $n = |\alpha|$; per analogia con le "fasi lunari", chiamiamo $A \rightarrow \bullet\alpha$ e $A \rightarrow \alpha\bullet$ rispettivamente la *fase nuova* e la *fase piena* della produzione. Per una produzione nulla $A \rightarrow \varepsilon$, la fase nuova e la fase piena coincidono; quest'unica fase verrà indicata con $A \rightarrow \bullet$.

Consideriamo a mo' d'esempio la grammatica ***G*** con produzioni:

$\mathtt{S} \rightarrow \mathtt{aSb} \mid \varepsilon$.

Le frasi di ***G*** sono stringhe della forma $\mathtt{a}^n\mathtt{b}^n$, $n \geq 0$, mentre le formule di ***G*** diverse dalle frasi sono tutte della forma $\mathtt{a}^n\mathtt{Sb}^n$, $n \geq 0$. Si osservi che ogni formula (e quindi ogni frase) di ***G*** ammette un'unica derivazione. Ad esempio, per la frase `aaaSbbb` abbiamo l'unica derivazione

(`S`, `aSb`, `aaSbb`, `aaaSbbb`)

Pertanto, tutte le formule di ***G*** sono formule destrorse. Le preformule di ***G*** sono `S`, ε ed ogni stringa di una delle seguenti tre forme: $\mathtt{a}^n$ ($n \geq 1$), $\mathtt{a}^n\mathtt{S}$ ($n \geq 1$) e $\mathtt{a}^n\mathtt{Sb}$ ($n \geq 1$); così, l'insieme delle preformule di ***G*** è dato $\{\mathtt{a}\}^*\{\mathtt{aSb}, \mathtt{S}, \varepsilon\}$.
Per ogni preformula di ***G*** diversa dal simbolo speciale, la tabella che segue riporta, tra l'altro, le fasi compatibili con la preformula.

preformula ($\lambda\beta$)	β	fase $A \rightarrow \beta\bullet\gamma$	$\lambda\beta\gamma$	*formula destrorsa*
ε	ε ε	$\mathtt{S} \rightarrow \bullet$ $\mathtt{S} \rightarrow \bullet\mathtt{aSb}$	ε $\mathtt{aSb}$	ε $\mathtt{aSb}$
$\mathtt{a}^n$ ($n \geq 1$)	ε ε $\mathtt{a}$	$\mathtt{S} \rightarrow \bullet$ $\mathtt{S} \rightarrow \bullet\mathtt{aSb}$ $\mathtt{S} \rightarrow \mathtt{a}\bullet\mathtt{Sb}$	$\mathtt{a}^n$ $\mathtt{a}^{n+1}\mathtt{Sb}$ $\mathtt{a}^n\mathtt{Sb}$	$\mathtt{a}^n\mathtt{b}^n$ $\mathtt{a}^{n+1}\mathtt{Sb}^{n+1}$ $\mathtt{a}^n\mathtt{Sb}^n$
$\mathtt{a}^n\mathtt{S}$ ($n \geq 1$)	$\mathtt{aS}$	$\mathtt{S} \rightarrow \mathtt{aS}\bullet\mathtt{b}$	$\mathtt{a}^n\mathtt{Sb}$	$\mathtt{a}^n\mathtt{Sb}^n$
$\mathtt{a}^n\mathtt{Sb}$ ($n \geq 1$)	$\mathtt{aSb}$	$\mathtt{S} \rightarrow \mathtt{aSb}\bullet$	$\mathtt{a}^n\mathtt{Sb}$	$\mathtt{a}^n\mathtt{Sb}^n$

Dunque, il nostro automa finito ***D*** sarà tale da accettare tutte e solo le preformule della grammatica, e ci riferiremo a ***D*** come all'*automa delle preformule*. A questo scopo, ogni stato q dell'automa finito ***D*** è esso stesso definito come l'insieme delle fasi

compatibili con le preformule che si ottengono considerando tutte le possibili sequenze di transizioni dallo stato iniziale q_0 di $\boldsymbol{D}$ allo stato q. (Si ricordi che ogni stato di $\boldsymbol{D}$ è di accettazione.) L'automa delle preformule $\boldsymbol{D}$ si ottiene partendo da un ε-AFN equivalente a $\boldsymbol{D}$. Vediamo, dunque, come è costruito questo ε-AFN.

Sia $\boldsymbol{G} = (N, T, P, S)$ la grammatica in esame. Innanzitutto, viene introdotta la *grammatica estesa*

$$\boldsymbol{G}' = (N', T, P', S')$$

dove

$$N' = N \cup \{S'\} \qquad P' = P \cup \{S' \rightarrow S\}.$$

Ora, a partire dalle fasi delle produzioni di $\boldsymbol{G}'$ si costruisce un ε-AFN $\boldsymbol{A}$ che ha:

— per stati le fasi delle produzioni di $\boldsymbol{G}'$,

— per alfabeto l'insieme $N \cup T$,

— per stato iniziale la fase $S' \rightarrow \bullet S$ della produzione $S' \rightarrow S$ di $\boldsymbol{G}'$,

— per stati di accettazione tutte le fasi delle produzioni di $\boldsymbol{G}'$.

Quanto alle transizioni di $\boldsymbol{A}$, ne avremo di spontanee e non. Ogni transizione nonspontanea sul simbolo X (appartenente a Σ)) ha come stato iniziale una qualsiasi fase del tipo $A \rightarrow \beta \bullet X \delta$ e come stato finale la fase $A \rightarrow \beta X \bullet \delta$. Ogni transizione spontanea ha come stato iniziale una fase del tipo $A \rightarrow \beta \bullet B \delta$ e come stato finale una fase del tipo $B \rightarrow \bullet \alpha$.

Vediamone ora un esempio con la grammatica $\boldsymbol{G}$ con produzioni

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{id}$

La grammatica estesa $\boldsymbol{G}'$ di $\boldsymbol{G}$ ha in più la produzione

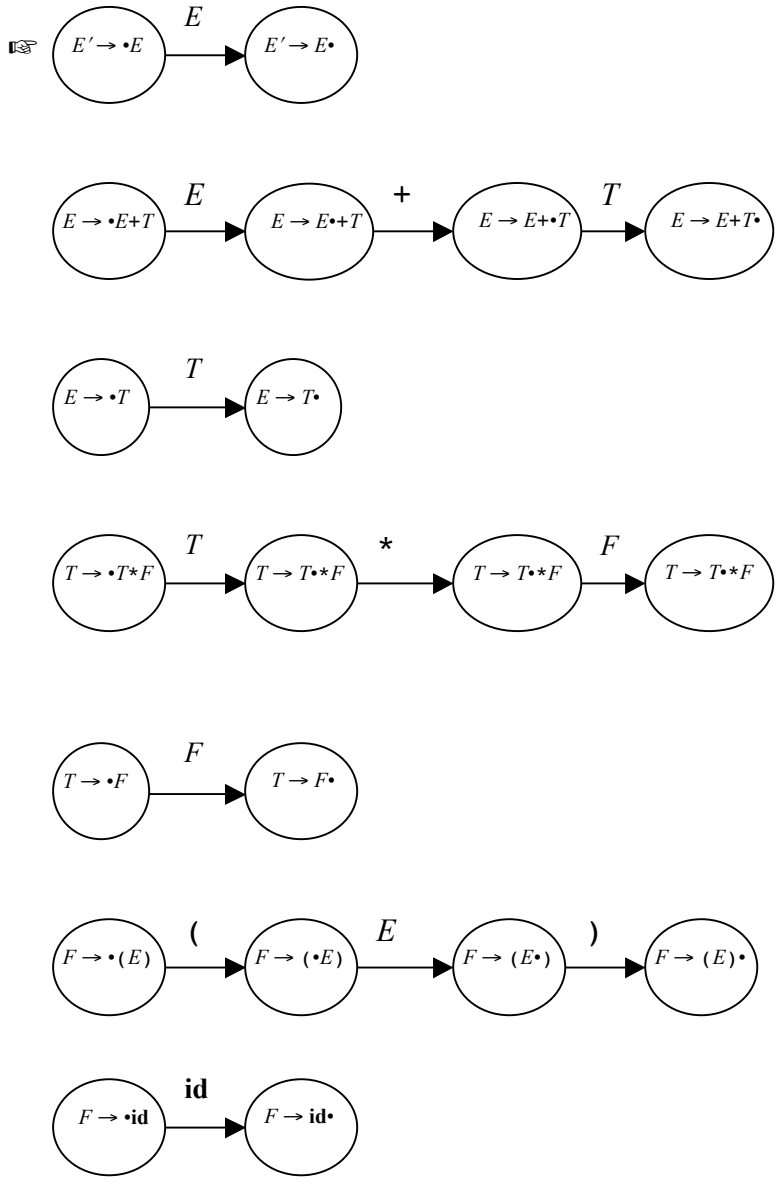
$$E' \rightarrow E$$

L'automa $\boldsymbol{A}$ ha allora 20 stati corrispondenti alle fasi delle produzioni di $\boldsymbol{G}'$:

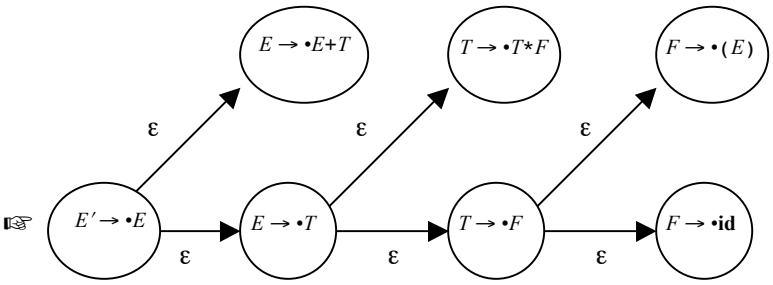
$E' \rightarrow \bullet E$ $\quad$ $E' \rightarrow E \bullet$
$E \rightarrow \bullet E + T$ $\quad$ $E \rightarrow E \bullet + T$ $\quad$ $E \rightarrow E + \bullet T$ $\quad$ $E \rightarrow E + T \bullet$

$E \to \bullet T$	$E \to T\bullet$		
$T \to \bullet T * F$	$T \to T\bullet * F$	$T \to T * \bullet F$	$T \to T * F\bullet$
$T \to \bullet F$	$T \to F\bullet$		
$F \to \bullet (E)$	$F \to (\bullet E)$	$F \to (E\bullet)$	$F \to (E)\bullet$
$F \to \bullet \mathbf{id}$	$F \to \mathbf{id}\bullet$		

Le transizioni nonspontanee sono



Qui di seguito sono anche riportate alcune delle transizioni spontanee di ***A***



Una volta completata la costruzione dell'automa ***A***, per ottenere l'automa delle preformule ***D***, applicheremo l'Algoritmo 8.2 ed elimineremo dall'AFD così ottenuto gli stati morti. Qui di seguito sono riportati gli stati di ***D***:

$q_0 = \{E' \rightarrow \bullet E,\ E \rightarrow \bullet E{+}T,\ E \rightarrow \bullet T,\ T \rightarrow \bullet T{*}F,\ T \rightarrow \bullet F,\ F \rightarrow \bullet(E),\ F \rightarrow \bullet\mathbf{id}\}$

$q_1 = \{E' \rightarrow E\bullet,\ E \rightarrow E\bullet{+}T\}$

$q_2 = \{E \rightarrow T\bullet,\ T \rightarrow T\bullet{*}F\}$

$q_3 = \{T \rightarrow F\bullet\}$

$q_4 = \{E \rightarrow \bullet E{+}T,\ E \rightarrow \bullet T,\ T \rightarrow \bullet T{*}F,\ T \rightarrow \bullet F,\ F \rightarrow \bullet(E),\ F \rightarrow (\bullet E),\ F \rightarrow \bullet\mathbf{id}\}$

$q_5 = \{F \rightarrow \mathbf{id}\bullet\}$

$q_6 = \{E \rightarrow E{+}\bullet T,\ T \rightarrow \bullet T{*}F,\ T \rightarrow \bullet F,\ F \rightarrow \bullet(E),\ F \rightarrow \bullet\mathbf{id}\}$

$q_7 = \{T \rightarrow T{*}\bullet F,\ F \rightarrow \bullet(E),\ F \rightarrow \bullet\mathbf{id}\}$

$q_8 = \{E \rightarrow E\bullet{+}T,\ F \rightarrow (E\bullet)\}$

$q_9 = \{E \rightarrow E{+}T\bullet,\ T \rightarrow T\bullet{*}F\}$

$q_{10} = \{T \rightarrow T{*}F\bullet\}$

$q_{11} = \{F \rightarrow (E)\bullet\}$

Per la tabella delle transizioni di ***D*** si veda alla pagina 118.

8.2.3 *Costruzione della tabella delle azioni*

Supponiamo che ad un certo istante il contenuto della pila sia (q_0, X_1, q_1, X_2, …, q_{m-1}, X_m, q_m) e che il contenuto del buffer sia y\$. Per quanto detto, la stringa $X_1X_2\ldots X_m$ contenuta nella pila è una preformula della grammatica e q_m contiene tutte le fasi $A \rightarrow \beta\bullet\gamma$ compatibili con $X_1X_2\ldots X_m$. Ora, se q_m contiene una fase piena $A \rightarrow \alpha\bullet$, allora potremmo applicare subito la produzione $A\rightarrow\alpha$ per ridurre $X_1X_2\ldots X_m$ e cioè eseguire l'azione R($A\rightarrow\alpha$); mentre, se q_m contiene una fase $A \rightarrow \beta\bullet\gamma$ con $\gamma \neq \varepsilon$, allora

dovremmo attendere per poter applicare la produzione $A \to \beta\gamma$ e quindi eseguire l'azione T. Ovviamente, due diverse fasi compatibili con $X_1X_2\ldots X_m$ potrebbero suggerire due azioni distinte e, talora, il dilemma può essere sciolto considerando il simbolo corrente, cioè il primo simbolo della stringa y\$ contenuta nel buffer.

Vediamo, dunque, come è costruita la tabella delle azioni, cioè qual è il contenuto della generica cella (q, e) dove q è uno stato dell'automa delle preformule ***D*** ed e è un elemento dell'insieme $T \cup \{\$\}$:

(1) Se q contiene una fase del tipo $A \to \beta \bullet e\gamma$ dove e è un simbolo terminale, allora la cella (q, e) conterrà (tra le altre) l'azione T.

(2) Se q contiene una fase piena $A \to \alpha\bullet$ con $A \neq S'$ e se $e \in \mathcal{J}(A)$, allora la cella (q, e) conterrà (tra le altre) l'azione R($A \to \alpha$).

(3) Se q contiene la fase piena $S' \to S\bullet$ ed e = \$, allora la cella $(q,$ \$) conterrà l'azione A ("accetta").

(4) In tutti gli altri casi, la cella (q, e) conterrà l'azione E ("errore").

A titolo illustrativo, riprendiamo la grammatica ***G*** con produzioni

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

e vediamo quali azioni che non siano E sono riportate nelle righe della tabella delle azioni corrispondenti ai tre stati dell'automa ***D***

$q_0 = \{E' \to \bullet E, E \to \bullet E+T, E \to \bullet T, T \to \bullet T*F, T \to \bullet F, F \to \bullet(E), F \to \bullet$**id**$\}$

$q_1 = \{E' \to E\bullet, E \to E\bullet+T\}$

$q_2 = \{E \to T\bullet, T \to T\bullet*F\}$

(q_0) La presenza della fase $F \to \bullet(E)$ fa sì che la cella (q_0, () contenga l'azione T, e la presenza della fase $F \to \bullet$**id** fa sì che la cella (q_0, **id**) contenga l'azione T. Le altre cinque fasi sono inefficaci, vale a dire, ogni cella (q_0, a) con $a \neq$ (oppure $a \neq$ **id** conterrà l'azione E.

(q_1) La presenza della fase piena $E' \to E\bullet$ fa sì che la cella (q_1, \$) contenga l'azione A e la presenza della fase $E \to E\bullet+T$ fa sì che la cella (q_1, +) contenga l'azione T.

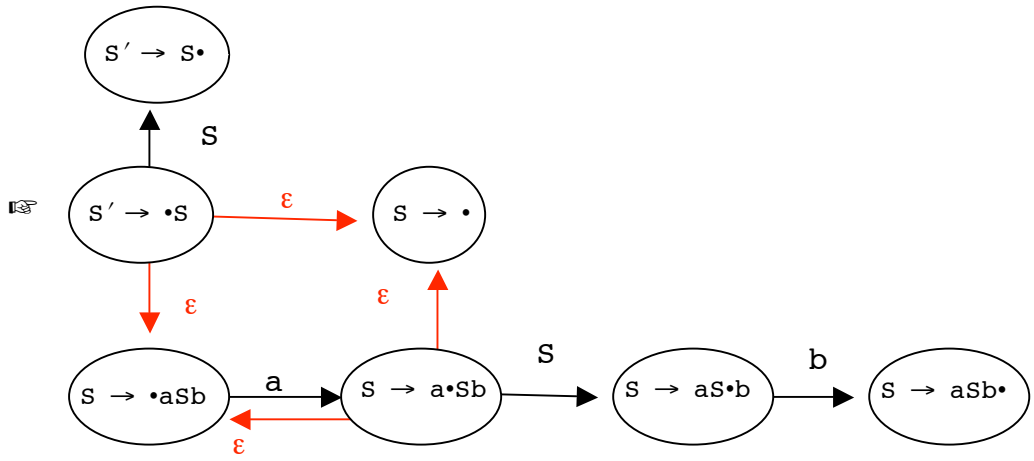
(q_2) Siccome $J(E) = \{\$, +,)\}$, la presenza della fase piena $E \rightarrow T\bullet$ fa sì che tutte e tre le celle (q_2, $), ($q_2$, +) ($q_2$,)) contengano l'azione R($E \rightarrow T$). La presenza della fase $T \rightarrow T\bullet * F$ fa sì che la cella (q_2, *) contenga l'azione T.

Le grammatiche (come quella in esame) per le quali sia possibile costruire una tabella delle azioni le cui celle contengano ciascuna una sola azione sono chiamate *grammatiche LR*(1) e con le grammatiche *LR*(1) la procedura di riduzione funziona correttamente, cioè senza le incertezze che potrebbero sorgere se una cella contenesse due o più azioni. Si osservi che, onde evitare incertezze è bene usare l'automa delle preformule nella maniera detta, senza cercare di minimizzarlo perché il risultato potrebbe portare ad una tabella delle azioni con celle multiple.

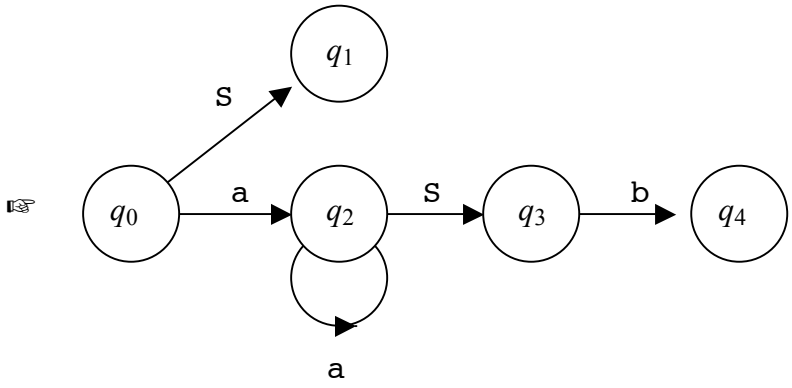
Torniamo infine a considerare la grammatica ***G*** con le due sole produzioni:

$$S \rightarrow aSb \mid \varepsilon.$$

Una volta costruito l'automa ***A***



otteniamo l'automa delle preformule ***D***:



dove

q_0 = {S′ → •S, S → •aSb, S → •}
q_1 = {S′ → S•}
q_2 = {S → •aSb, S → a•Sb, S → •}
q_3 = {S → aS•b }
q_4 = {S → aSb•}

Dopo aver calcolato J(S) = {b, $}, otteniamo la tabella delle azioni:

stato	a	b	$
q_0	T	R (S → ε)	R (S → ε)
q_1			A
q_2	T	R (S → ε)	R (S → ε)
q_3		T	
q_4		R (S → aSb)	R (S → aSb)

le cui celle sono tutte semplici, la qual cosa dimostra che ***G*** appartiene alla classe *LR*(1).

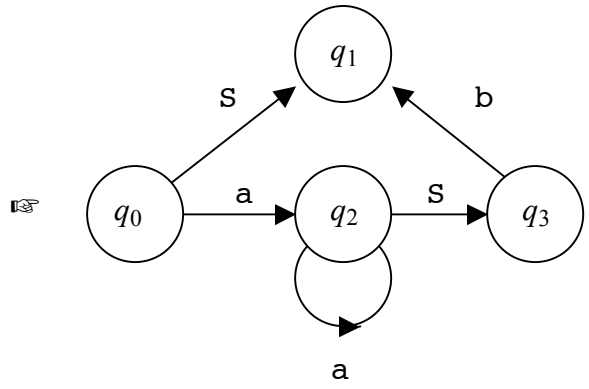
Se applichiamo la procedura di riduzione alla frase aabb di ***G*** otteniamo

pila	*buffer*	*azione*
q_0	aabb$	T
q_0aq_2	abb$	T
q_0aq_2aq_2	bb$	R (S → ε)
q_0aq_2aq_2Sq_3	bb$	T
q_0aq_2aq_2Sq_3bq_4	b$	R (S → aSb)
q_0aq_2Sq_3	b$	T
q_0aq_2Sq_3bq_4	$	R (S → aSb)
q_0Sq_1	$	A

Se invece applichiamo la procedura di riduzione alla stringa aab otteniamo

pila	*buffer*	*azione*
q_0	aab$	T
q_0aq_2	ab$	T
q_0aq_2aq_2	b$	R (S → ε)
q_0aq_2aq_2Sq_3	b$	T
q_0aq_2aq_2Sq_3bq_4	$	R (S → aSb)
q_0aq_2Sq_3	$	E

Si osservi infine che un automa equivalente a ***D*** di dimensione minima si ottiene fondendo assieme i due stati indistinguibili q_1 e q_4.



Ma così la tabella delle azioni che ne risulta

stato	a	b	$
q_0	T	R (S → ε)	R (S → ε)
q_1		R (S → aSb)	A R (S → aSb)
q_2	T	R (S → ε)	R (S → ε)
q_3		T	

viene a contenere una cella doppia.

Cap. 9

ANALISI SEMANTICA

Fino a questo punto, del linguaggio sorgente è stata usata solo la sua componente formale che ne definisce il lessico e la sintassi, così che il testo è stato esaminato come fosse una qualsiasi stringa di caratteri. Solo ora entra in gioco la *semantica* del linguaggio sorgente, che fa del testo un programma di calcolo. La semantica del linguaggio sorgente viene specificata usando un approccio *operazionale* che associa a certi simboli grammaticali (terminali e non) un *attributo* (talora specificato come una funzione del simbolo grammaticale) e ad ogni produzione della grammatica della sintassi una *regola semantica*, cioè un insieme di operazioni sugli attributi associati ai simboli grammaticali coinvolti nella produzione.

Esempio 9.1. Consideriamo il linguaggio di programmazione di una semplice macchina calcolatrice da tavolo che esegue operazioni di addizione e moltiplicazione di numeri naturali. Per semplicità soprassediamo all'introduzione di costanti lessicali per i lessemi `+`, `*`, `(` e `)` ed utilizziamo la grammatica $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}, \mathtt{E}, \mathtt{T}, \mathtt{F}\}$, $T = \{\mathbf{num}, \mathtt{+}, \mathtt{*}, \mathtt{(}, \mathtt{)}\}$ e P consta delle sette produzioni

P1: `S` → `E`
P2: `E` → `E + T`
P3: `E` → `T`
P4: `T` → `T * F`
P5: `T` → `F`
P6: `F` → `(E)`
P7: `F` → **num**

La semantica del linguaggio è definita associando ai simboli grammaticali `E`, `T`, `F` e **num** gli attributi *val*(`E`), *val*(`T`), *val*(`F`) e *val*(**num**), che sono variabili di tipo intero nonnegativo, e alle sette produzioni le seguenti regole semantiche.

produzione		*regola semantica*	
P1:	`S` → `E`	R1:	*stampare val*(`E`)
P2:	`E` → $\mathtt{E}_1$ `+ T`	R2:	*val*(`E`) := *val*($\mathtt{E}_1$) + *val*(`T`)
P3:	`E` → `T`	R3:	*val*(`E`) := *val*(`T`)
P4:	`T` → $\mathtt{T}_1$ `* F`	R4:	*val*(`T`) := *val*($\mathtt{T}_1$) * *val*(`F`)
P5:	`T` → `F`	R5:	*val*(`T`) := *val*(`F`)
P6:	`F` → `(E)`	R6:	*val*(`F`) := *val*(`E`)
P7:	`F` → **num**	R7:	*val*(`F`) := *val*(**num**)

Si osservi che, se un simbolo nonterminale A appare più di una volta in una stessa produzione (v. produzioni P2 e P4), allora per non creare ambiguità la sua seconda (terza, …) occorrenza è indicata con A_1 (A_2, …).

9.1 Interpretazione

Consideriamo un testo tale che il suo schema lessicale $\mathbf{t}_1 \ldots \mathbf{t}_k$ sia una frase della grammatica della sintassi. Supponiamo di aver costruito un suo albero di derivazione $\mathcal{T}$. Così, le foglie di $\mathcal{T}$ sono etichettate nell'ordine dalle figure lessicali $\mathbf{t}_1$, …, $\mathbf{t}_k$. Il primo passo per determinare il significato operazionale (programma) del testo prevede l'*interpretazione* dei vertici di $\mathcal{T}$ sulla base delle regole semantiche e dello schema annotato, sia esso ($\mathbf{t}_1$: n_1), …, ($\mathbf{t}_k$: n_k). Esplicitamente,

— per ogni vertice u di $\mathcal{T}$ etichettato con una variabile, sia essa A, detti v_1, …, v_m i vertici che sono figli di u, l'interpretazione di u è data dalla regola semantica associata alla produzione

$$A \rightarrow X_1 X_2 \ldots X_m$$

dove X_i è il simbolo grammaticale che etichetta v_i ($1 \leq i \leq m$).

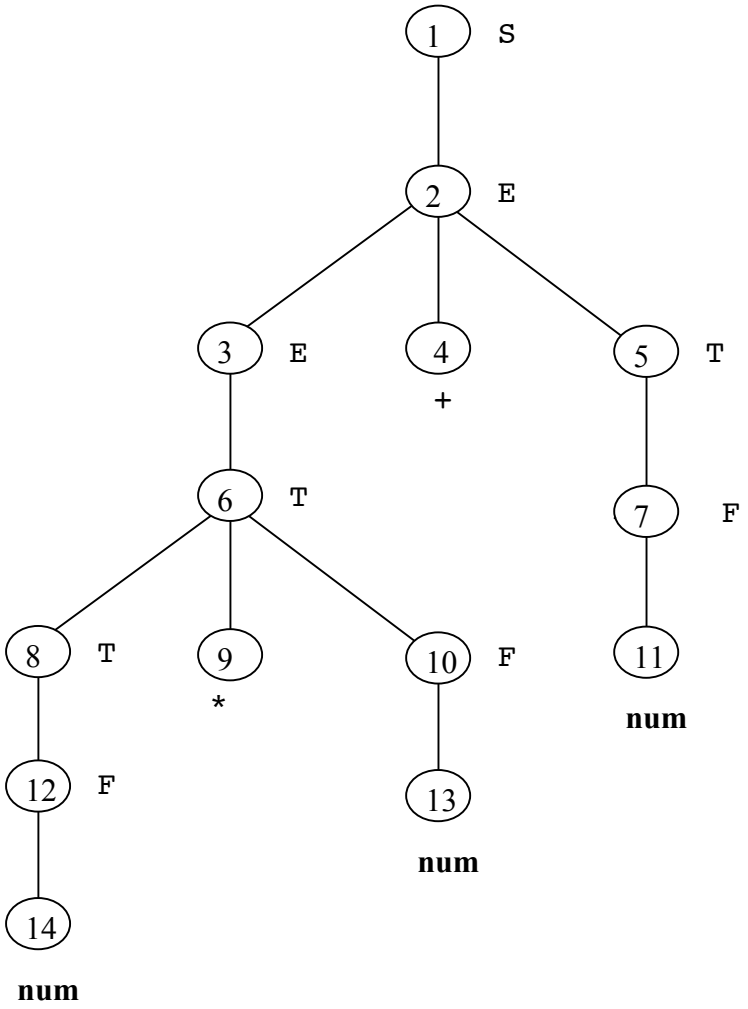
— per ogni foglia u di $\mathcal{T}$ etichettata con una figura lessicale che non sia una costante lessicale, l'interpretazione di u è un'appropriata istruzione che opera sull'etichetta di u.

Chiamiamo l'albero che così si ottiene da $\mathcal{T}$ l'*albero di derivazione interpretato*.

Esempio 9.1 (seguito). Consideriamo ad esempio la stringa `30*5+4`. Il suo schema lessicale ed il suo schema annotato sono rispettivamente

num * num + num (**num**: `30`) * (**num**: `5`) + (**num**: `4`)

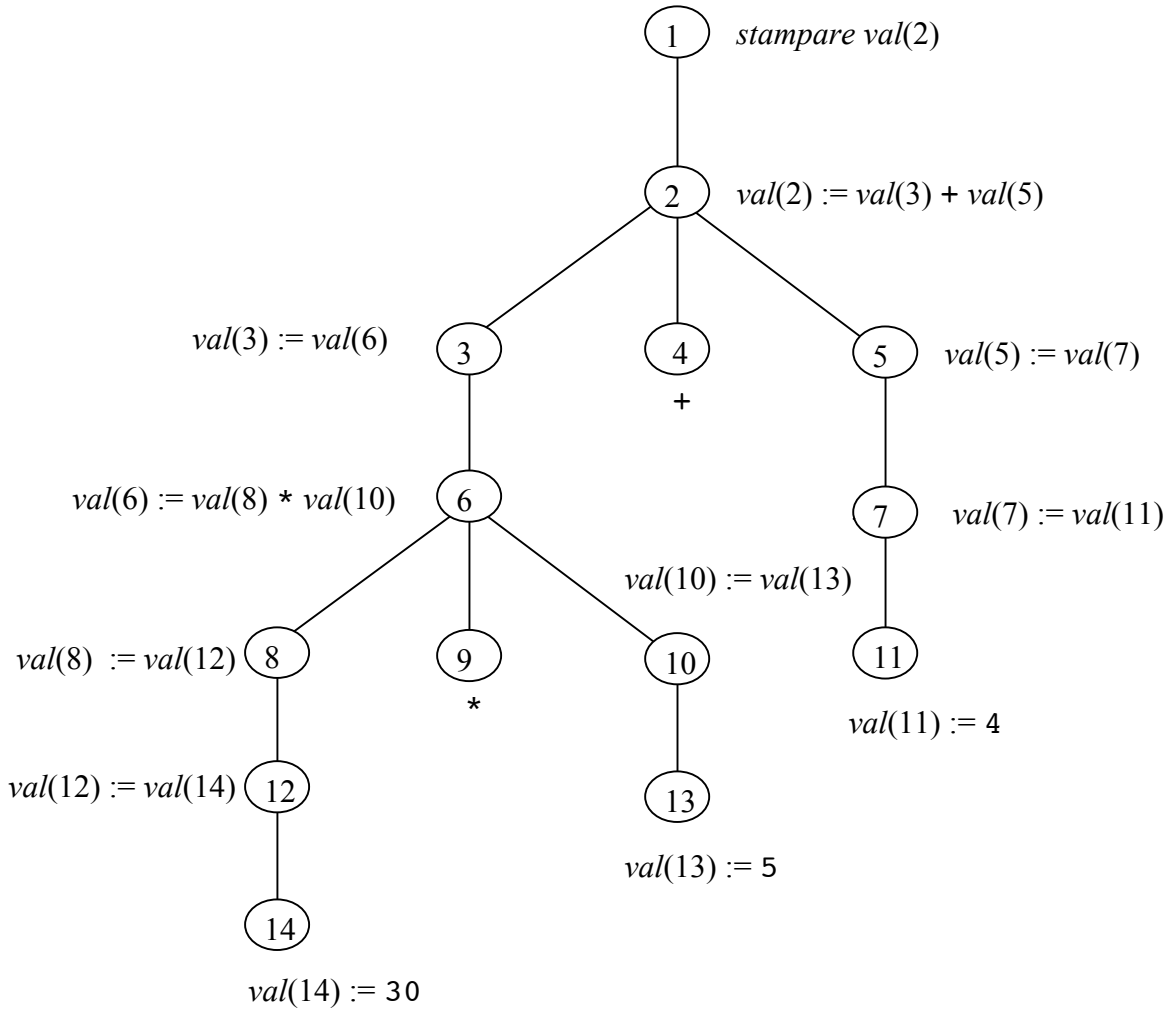
Lo schema lessicale è una frase di $\boldsymbol{G}$ ed il suo albero di derivazione è mostrato in figura.



Consideriamo la radice dell'albero. Il vertice 1 ha etichetta `S` ed ha un figlio, il vertice 2 etichettato con `E`. Pertanto, l'interpretazione del vertice 1 è data dalla regola semantica associata alla produzione `S` → `E`, cioè '*stampare val*(`E`)', che riscriviamo come

'*stampare val*(2)'.

Consideriamo poi il vertice 2. Questo ha etichetta `E` ed ha tre figli, nell'ordine i vertici 3, 4 e 5, etichettati rispettivamente con `E`, con `+` e con `T`. Pertanto l'interpretazione del vertice 2 è data dalla regola semantica associata alla produzione `E` → $\texttt{E}_1$ `+` `T`, cioè *val*(`E`) := *val*($\texttt{E}_1$) + *val*(`T`), che riscriviamo come *val*(2) := *val*(3) + *val*(5). E così via fino a raggiungere le foglie. Consideriamo ora le foglie, ad esempio, il vertice 14 etichettato con **num**. Essendo (**num**: `30`) la sua registrazione, l'interpretazione del vertice 14 è data dall'istruzione *val*(**num**) := `30`, che riscriviamo come *val*(14) := `30`. Dunque, l'albero di derivazione interpretato è quello mostrato in figura.



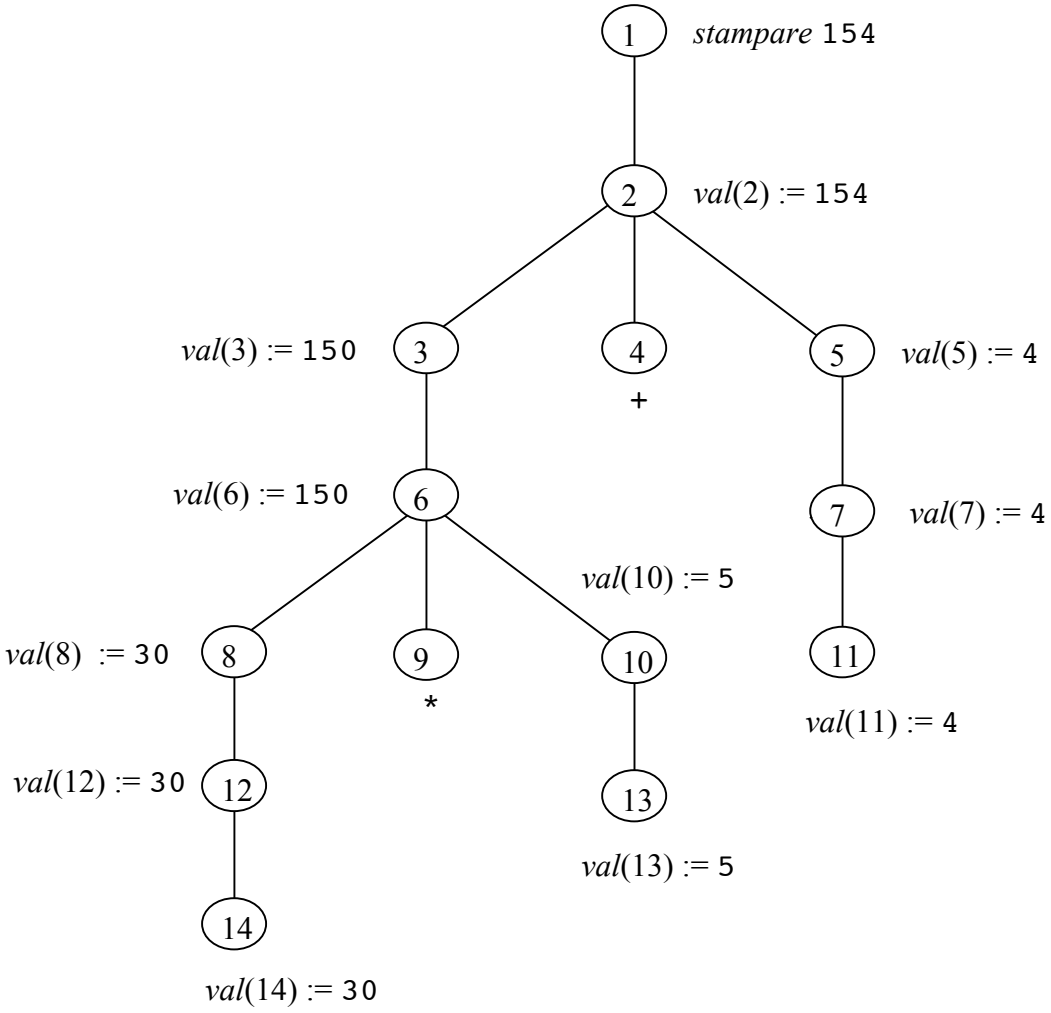
9.2 Traduzione

Come mostrato nell'esempio precedente, tra gli attributi associati ai simboli grammaticali, nella fattispecie *val*(2), …, *val*(14), esistono delle relazioni di dipendenza; ad esempio, *val*(2) dipende da *val*(3) e da *val*(5) cosicché *val*(2) non può essere calcolato se non dopo *val*(3) e *val*(5). Se elenchiamo le interpretazioni (cioè le istruzioni) dei vertici dell'albero di derivazione interpretato rispettando l'*ordine di precedenza* imposto dalle relazioni di dipendenza, otteniamo una lista di istruzioni, che chiamiamo una *traduzione* del programma, e la loro esecuzione fornirà il significato operazionale del programma.

Esempio 9.1 (seguito). L'ordine di precedenza imposto dalle relazioni di dipendenza è tale che, se *v* è figlio di *u*, allora *val*(*v*) va calcolato prima di *val*(*u*). Così, una traduzione del programma è la seguente

```
val(14) := 30;
val(13) := 5;
val(12) := val(14);
val(11) := 4;
val(10) := val(13);
val(8)  := val(12);
val(7) := val(11);
val(6) := val(8) * val(10);
val(5) := val(7);
val(3) := val(6);
val(2) := val(3) + val(5);
stampare val(2).
```

e la sua esecuzione è illustrata dall'albero mostrato in figura



Esempio 9.2 Consideriamo le istruzioni di dichiarazione di tipo per gli identificatori di un certo linguaggio di programmazione. Assumiamo che la grammatica generativa di tali istruzioni sia $\boldsymbol{G} = (N, T, P, \mathtt{S})$, dove $N = \{\mathtt{S}, \mathtt{T}, \mathtt{I}\}$, $T = \{$**int**, **real**, **id**, **,**$\}$ e P consta delle cinque produzioni

P1: $\mathtt{S} \rightarrow \mathtt{T}\ \mathtt{I}$
P2: $\mathtt{T} \rightarrow$ **int**
P3: $\mathtt{T} \rightarrow$ **real**
P4: $\mathtt{I} \rightarrow \mathtt{I}$ **,** **id**
P5: $\mathtt{I} \rightarrow$ **id**

La semantica del linguaggio è definita associando ai simboli grammaticali `I`, `T` e **id** gli attributi *tipo*(`I`), *tipo*(`T`) e *tipo*(**id**), che sono variabili definite sull'insieme {*integer*, *real*} e alle cinque produzioni le seguenti regole semantiche.

produzione		*regola semantica*	
P1:	$\mathtt{S} \rightarrow \mathtt{T}\ \mathtt{I}$	R1:	*tipo*(`I`) := *tipo*(`T`)
P2:	$\mathtt{T} \rightarrow$ **int**	R2:	*tipo*(`T`) := *integer*
P3:	$\mathtt{T} \rightarrow$ **real**	R3:	*tipo*(`T`) := *real*
P4:	$\mathtt{I} \rightarrow \mathtt{I}_1$ **,** **id**	R4:	*tipo*($\mathtt{I}_1$) := *tipo*(`I`); *tipo*(**id**) := *tipo*(`I`)
P5:	$\mathtt{I} \rightarrow$ **id**	R5:	*tipo*(**id**) := *tipo*(`I`)

Consideriamo la dichiarazione di tipo

```
real e, m, c
```

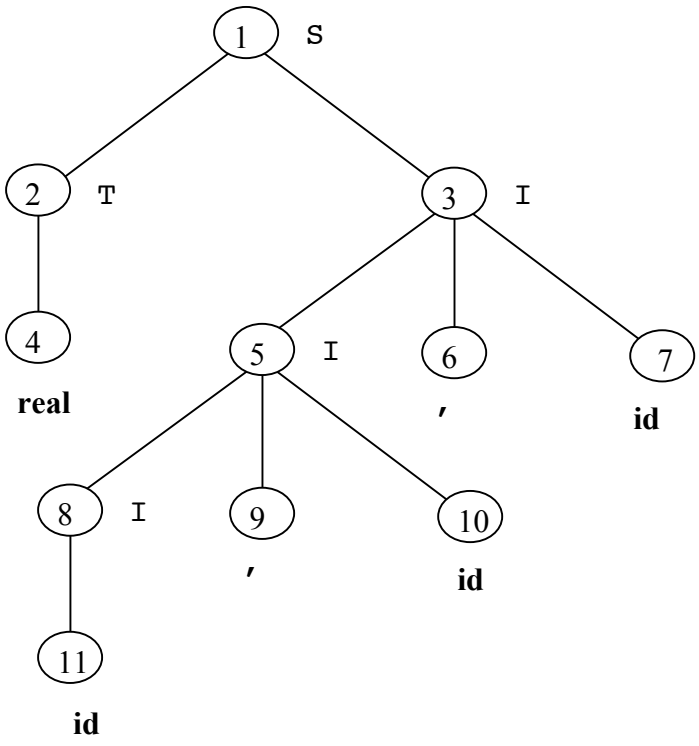
Il suo schema lessicale è

real id , id , id

ed il suo schema annotato è

real (**id**: 20) **,** (**id**: 21) **,** (**id**: 22)

Lo schema lessicale è una frase di $\boldsymbol{G}$ e l'albero di derivazione è mostrato in figura.



Consideriamo la radice dell'albero. Il vertice 1 ha etichetta `S` ed ha due figli, il vertice 2 etichettato con `T` ed il vertice 3 etichettato con `I`. Pertanto l'interpretazione del vertice 1 è data dalla regola semantica associata alla produzione `S` → `T I`, cioè *tipo*(`I`) := *tipo*(`T`), che riscriviamo come

tipo(3) := *tipo*(2).

Consideriamo poi il vertice 2. Questo ha un figlio, il vertice 4 etichettato con **real**. Pertanto l'interpretazione del vertice 2 è data dalla regola semantica associata alla produzione `T` → **real**, cioè *tipo*(`T`) := *real*, che riscriviamo come

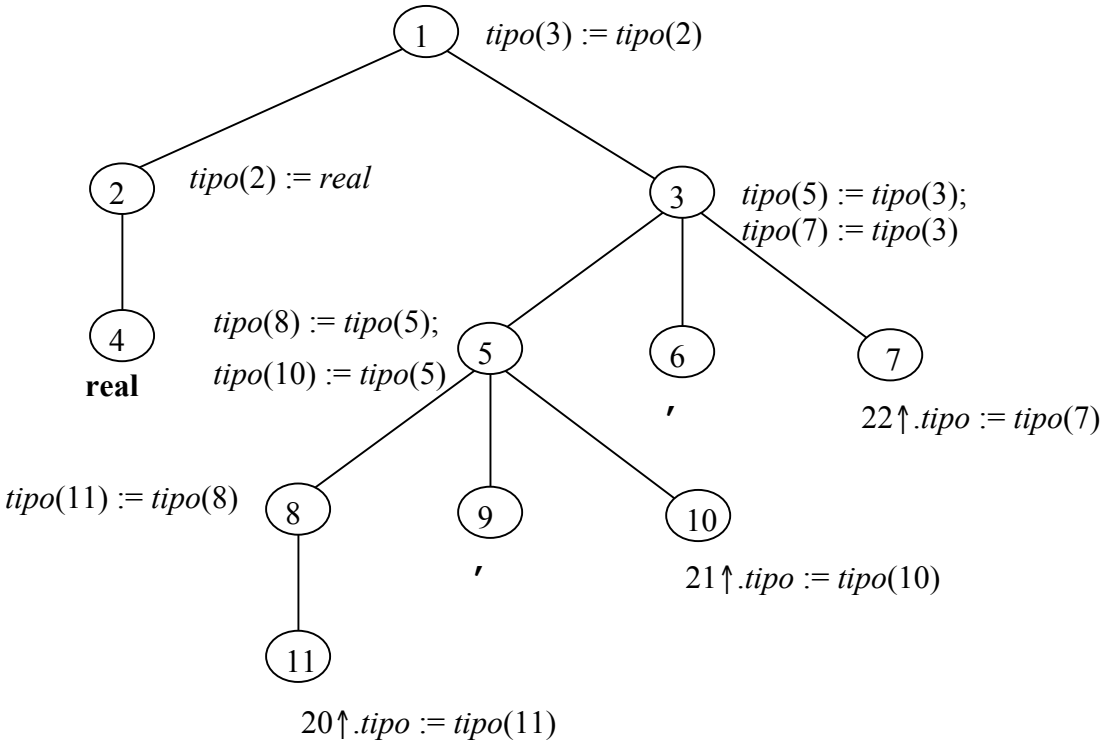
tipo(2) := *real*.

E così via fino a raggiungere le foglie. Consideriamo ora il vertice 11 etichettato con **id**. Essendo (**id**: 20) la sua registrazione, l'interpretazione del vertice 11 è data dall'istruzione

20↑.*tipo* := *tipo*(11)

che ha l'effetto di copiare *tipo*(11) nel campo *tipo* della riga 20 della tabella dei simboli.

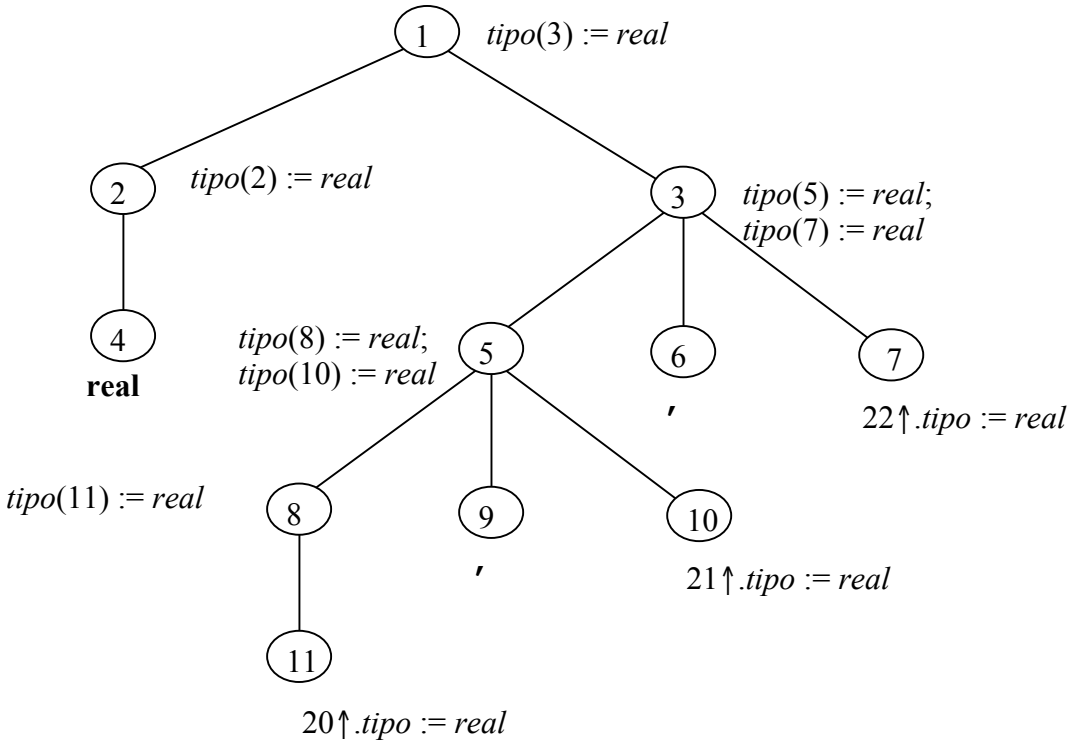
Dunque, l'albero di derivazione interpretato è



ed una traduzione è la seguente

tipo(2):= *real*;
tipo(3) := *tipo*(2);
tipo(5) := *tipo*(3);
tipo(7) := *tipo*(3);
tipo(8) := *tipo*(5);
tipo(10) := *tipo*(5);
22↑.*tipo* := *tipo*(7);
tipo(11) := *tipo*(8);
21↑.*tipo* := *tipo*(10);
20↑.*tipo* := *tipo*(11).

L'albero che segue ne illustra il processo di esecuzione



ed il risultato consiste nello scrivere *real* nel campo *tipo* delle righe 20, 21 e 22 della tabella dei simboli:

	lessema	*figura*	*tipo*
1	`program`	**pgm**	…
	…	**…**	…
19	`and`	**and**	…
20	`e`	**id**	*real*
21	`m`	**id**	*real*
22	`c`	**id**	*real*

A questo punto sarà facile effettuare i controlli di tipo degli identificatori, che è uno dei compiti dell'analisi semantica.

9.3 L'albero della sintassi

Per ottenere la traduzione di un programma più spesso si utilizza, anziché l'albero di derivazione interpretato, una sua rappresentazione operazionale (l'*albero della sintassi*) la quale viene costruita utilizzando regole semantiche appropriate, dette *regole grafiche*. Per semplicità, mostreremo come costruire l'albero della sintassi di un'espressione del linguaggio sorgente.

L'albero della sintassi di un'espressione è un albero ordinato, i cui vertici rappresentano sottoespressioni e sono etichettati con operatori (vertici-operatori) o con operandi (vertici-operandi); inoltre, i figli di un vertice-operatore *u* rappresentano le sottoespressioni che formano l'espressione rappresentata da *u*. Ogni vertice dell'albero della sintassi corrisponde (univocamente) ad un record ed il record corrispondente ad un vertice-operatore *u* ha un campo destinato all'etichetta del vertice (un operatore) mentre gli altri campi contengono puntatori ai record corrispondenti ai figli di *u*. La costruzione dell'albero della sintassi è il risultato dell'interpretazione dell'albero di derivazione per lo schema lessicale mediante regole grafiche basate sulle funzioni grafiche *Nodo* e *Foglia*:

1. *Nodo*(*op*: *primo*, *secondo*) crea un vertice-operatore con etichetta *op* e due puntatori ai record corrispondenti ai suoi figli;
2. *Foglia*(**id**: *n*) crea un vertice-operando con etichetta **id** ed un puntatore (*n*) ad una riga della tabella dei simboli;
3. *Foglia*(**num**: *x*) crea un vertice-operando con etichetta **num** ed un numero *x*.

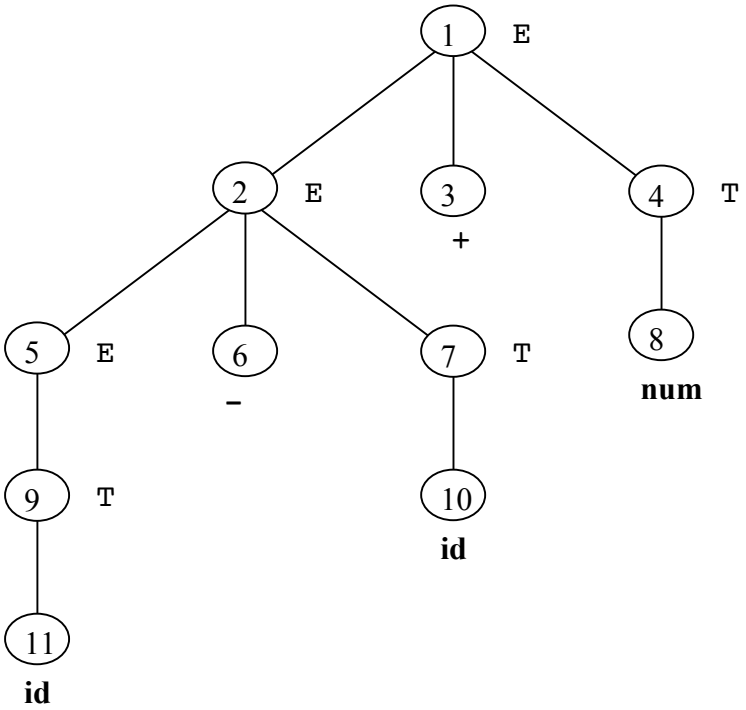
Esempio 9.3 Consideriamo le seguenti produzioni con le associate regole grafiche:

produzione		*regola grafica*	
P1:	`E` → `E`$_1$ `+ T`	G1:	*p*(`E`) := *Nodo*(`+`: *p*(`E`$_1$), *p*(`T`))
P2:	`E` → `E`$_1$ `- T`	G2:	*p*(`E`):= *Nodo*(`-`: *p*(`E`$_1$), *p*(`T`))
P3:	`E` → `T`	G3:	*p*(`E`):= *p*(`T`)
P4:	`T` → `(E)`	G4:	*p*(`T`) := *p*(`E`)
P5:	`T` → **id**	G5:	*p*(`T`) := *Foglia*(**id**: *n*)
P6:	`T` → **num**	G6:	*p*(`T`) := *Foglia*(**num**: *x*)

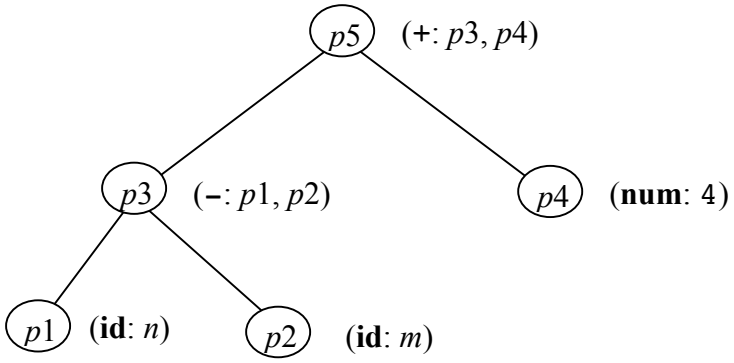
e l'espressione

```
a - b + 4
```

Il suo schema lessicale **id – id + num** è una frase della grammatica con albero di derivazione



L'albero della sintassi è costruito interpretando l'albero di derivazione a partire dalle foglie e procedendo fino alla radice:



Qui, *n* e *m* sono i puntatori alle righe della tabella dei simboli che contengono rispettivamente gli identificatori `a` e `b`. La traduzione dell'espressione in esame è allora data dalla lista delle seguenti istruzioni

*p*1 := *Foglia*(**id**: *n*)
*p*2 := *Foglia*(**id**: *m*)
*p*3 := *Nodo*(**–**: *p*1, *p*2)
*p*4 := *Foglia*(**num**: 4)
*p*5 := *Nodo*(**+**: *p*3, *p*4)

che sono nella forma *a tre indirizzi*, come richiesto dal codice intermedio (v. paragrafo 2.2).

Appendice

Sia V un insieme finito ed A un insieme di coppie ordinate di elementi di V. Si chiama *grafo orientato* l'oggetto matematico definito dalla coppia $\mathcal{D} = (V, A)$; esso viene rappresentato disegnando un punto (chiamato *vertice*) per ogni elemento di V ed una freccia (chiamato *arco*) dal vertice u al vertice v per ogni coppia (u, v) appartenente ad A. Se (u, v) è un arco del grafo, allora esso è un *arco entrante* per il vertice v ed un *arco uscente* dal vertice u.

Un *cammino* di *lunghezza* k ($k \geq 0$) è una successione di vertici $(v_0, v_1, \ldots, v_k)$ tale che se $k > 0$ allora, per ogni $h < k$, (v_h, v_{h+1}) o (v_{h+1}, v_h) è un arco; i vertici v_0 e v_k sono chiamati gli *estremi* del cammino $(v_0, v_1, \ldots, v_k)$, che prende il nome di cammino *tra* v_0 *a* v_k. Un *ciclo* (o circuito) è un cammino di lunghezza $k > 0$ i cui estremi coincidono.

Due vertici u e v sono *connessi* se esiste un cammino tra u a v. La connessione tra due vertici è una relazione di equivalenza. Un grafo orientato è *connesso* se, per ogni coppia u e v di vertici, u e v sono connessi.

Un *cammino orientato* di *lunghezza* k ($k \geq 0$) è una successione di vertici $(v_0, v_1, \ldots, v_k)$ tale che se $k > 0$ allora, per ogni $h < k$, (v_h, v_{h+1}) è un arco; i vertici v_0 e v_k sono chiamati gli *estremi* del cammino orientato $(v_0, v_1, \ldots, v_k)$, che prende il nome di cammino *da* v_0 *a* v_k. Un *ciclo* (o circuito) *orientato* è un cammino orientato di lunghezza $k > 0$ i cui estremi coincidono.

Un vertice v è *raggiungibile* da un vertice u se esiste un cammino orientato da u a v. La raggiungibilità è una relazione tra vertici che gode della proprietà riflessiva e transitiva. Due vertici u e v sono *fortemente connessi* se v è *raggiungibile* da u o u è *raggiungibile* da v. Un grafo orientato è *fortemente connesso* se, per ogni coppia u e v di vertici, u e v sono fortemente connessi.

Glossario inglese

albero della sintassi	syntax tree
albero di derivazione	parse tree
analisi sintattica	parsing
corsa	lookhead
derivazione destrorsa	rightmost derivation
derivazione sinistrorsa	leftmost derivation
fase	item
figura lessicale	token
formula	sentential form
frase	sentence
grafo orientato	directed graph (o digraph)
grammatica a struttura sintagmatica	phrase-structured grammar
grammatica acontestuale	context-free grammar
grammatica contestuale	context-sensitive grammar
grammatica della sintassi	syntax grammar
grammatica regolare	regular grammar
lessema	lexeme
metodo deduttivo	top-down method
metodo induttivo	bottom-up method
oggetto della riduzione	handle
preformula	viable prefix
produzione ricorsiva a sinistra	left-recursive production
programma di caricamento	loader
programma di collegamento	link editor
"raccoglimento a sinistra del fattor comune"	left factoring
riduzione destrorsa	left-rightmost reduction
scansione	scanning
schema lessicale	token stream
sentinella	sentinel
sintagma	phrase
stato iniziale (o *di partenza*)	start state
tabella dei simboli	symbol table
"tornare sui propri passi"	to backtrack