Michele Di Santo Nota n. 01

LaP06 2 gennaio 2006

Linguaggi, grammatiche, automi e traduzione diretta da sintassi

LINGUAGGI E GRAMMATICHE

Le prime definizioni

Sia T un alfabeto finito di simboli (spesso caratteri). Si definisce **linguaggio** su T un qualsiasi sottoinsieme dell'insieme T* di tutte le stringhe (sequenze) di simboli di T. Le **stringhe** del linguaggio L prendono il nome di **sentenze**[1] di L.

È ovvio dalla definizione che, dato un linguaggio L su T, in generale, *non tutti gli elementi di T* sono sentenze di L*.

Una **grammatica** G di L è un insieme di regole sintattiche che consentono di generare le sole sentenze di L, ovvero di verificare se un dato elemento di T* è oppure no una sentenza di L.

In generale, se G è una grammatica definita sull'insieme di simboli T, con L(G) si denota il linguaggio da essa definito, ossia il sottoinsieme di T* i cui elementi sono riconosciuti da G come sentenze di L.

Linguaggi e grammatiche liberi da contesto

Le grammatiche più utilizzate per la definizione delle regole sintattiche dei linguaggi di programmazione sono le **grammatiche libere da contesto** (context-free) [2].Una grammatica libera da contesto G per il linguaggio L è definita come una quadrupla <T, N, S, P>, in cui:

1. T è l'alfabeto dei simboli terminali del linguaggio L.
2. N è l'alfabeto dei simboli non terminali, ciascuno dei quali specifica una classe o categoria sintattica.
3. S è il simbolo non terminale (S $\in$ N) detto **simbolo di start**.
4. P è l'insieme delle **produzioni** o equazioni sintattiche ciascuna delle quali ha la forma:

   $$X = U$$

   dove X $\in$ N è un simbolo nonterminale, U $\in$ (T $\cup$ N)* è una stringa di simboli terminali e non terminali e il segno = è letto "può essere sostituito da". La stringa U può anche essere vuota, ossia avere lunghezza 0; in questo caso si indica con ε. Ovviamente, per ogni simbolo non terminale X, deve esistere in P almeno una produzione avente X alla sinistra del segno =.

La dizione "libera da contesto" sta a significare che, per ogni produzione di G, la sostituzione del simbolo X con la stringa U è sempre possibile, indipendentemente dal contesto in cui il simbolo X è utilizzato. Una **derivazione** è l'applicazione di una singola produzione, con sostituzione del simbolo non terminale a sinistra del segno = con la stringa a destra.

[1] Se L è un linguaggio di programmazione, le sentenze di L prendono il nome di programmi.

[2] Non tutti i linguaggi sono definibili usando grammatiche libere da contesto. In base alla classificazione di Chomsky, le grammatiche libere da contesto appartengono al livello 2, con i livelli 1 e 0 includenti rispettivamente le più generali grammatiche sensibili al contesto (context-sensitive) e non ristrette (unrestricted).

Data la grammatica G = <T, N, S, P>, il linguaggio L(G) è il sottoinsieme di T* i cui elementi possono essere generati, a partire dal simbolo di start S, tramite la ripetuta applicazione delle produzioni in P. Ciascun elemento di L(G) è una sentenza di L.

Ad esempio, il linguaggio delle stringhe formate da 0 o più caratteri **a**: L = {ε, **a**, **aa**, **aaa**, **aaaa**, …} può essere generato da una grammatica con le seguenti produzioni:

S = ε
S = **a** S

Notiamo come l'uso della **ricorsività** consenta di generare un linguaggio infinito usando un numero finito di produzioni. In particolare, la **ricorsività diretta** (X = … X …) è utile per esprimere un numero illimitato di ripetizioni.

Come ulteriore esempio, consideriamo il linguaggio delle espressioni che impiegano gli operandi **a**, **b**, **c** e **d**, gli operatori **+** e *****, con la moltiplicazione avente precedenza sull'addizione, e le parentesi **(** e **)**. Questo linguaggio può essere definito dalla grammatica le cui produzioni sono:

E = T	F = **(** E **)**
E = E **+** T	V = **a**
T = F	V = **b**
T = T ***** F	V = **c**
F = V	V = **d**

Ovviamente, T è l'insieme {a, b, c, d, (,), +, *}, N è l'insieme {E, T, F, V} ed E è il simbolo di start. I simboli non terminali definiscono le **classi** o **categorie sintattiche** delle espressioni (E), dei termini (T), dei fattori (F) e delle variabili (V).

Al fine di rendere più compatta la notazione, si preferisce di solito riunire tutte le produzioni aventi lo stesso simbolo non terminale a sinistra del segno di = in un'unica produzione che fa uso del metasimbolo | , da leggere "oppure", per separare le diverse alternative:

E = T | E **+** T
T = F | T ***** F
F = V | **(** E **)**
V = **a** | **b** | **c** | **d**

Questa notazione prende il nome di **BNF**, *Backus Naur Form*, dal nome dei due studiosi che l'hanno proposta.

Ulteriori abbreviazioni portano alla cosiddetta **EBNF** (*Extended BNF*) che utilizza come metasimboli le parentesi quadre [ ] per indicare **opzionalità** (0 o 1 volta) e le parentesi graffe { } per indicare **ripetizione** (0 o più volte). Inoltre, le parentesi tonde ( ) possono essere usate per raggruppare sottostringhe di simboli. Esprimendo la sintassi nella EBNF, non è più necessario usare il simbolo ε per esprimere la stringa vuota. Ad esempio, il linguaggio delle stringhe di **a** può esser generato dalla grammatica avente la produzione:

S = { **a** }

Analogamente, le produzioni della grammatica del linguaggio dell'espressioni si scrivere in EBNF:

E = T { **+** T }
T = F { ***** F }
F = V | **(** E **)**
V = **a** | **b** | **c** | **d**

L'uso della EBNF semplifica l'espressione di una grammatica, in quanto riduce l'impiego di produzioni ricorsive. Notiamo che la grammatica precedente fa ancora uso di **ricorsività indiretta**, in quanto la produzione relativa alla classe sintattica F usa a destra la classe sintattica E.

In generale, dato un linguaggio L, esistono più grammatiche in grado di generarlo. Ad esempio, il linguaggio delle stringhe contenenti un numero dispari di caratteri **a** può essere generato da una qualsiasi delle seguenti tre produzioni:

S = **a** | S **a a** S = **a** | **a** S **a** S = **a** | **a a** S

Più semplicemente, in EBNF si ha:

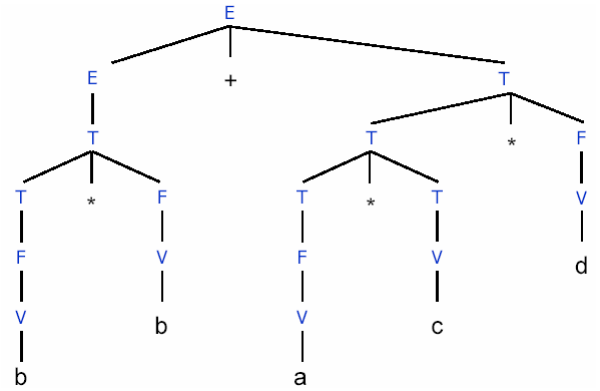
S = **a** { **a a** } S = { **a a** } **a**

Alberi sintattici e grammatiche ambigue

Una grammatica libera da contesto G determina un insieme di **alberi sintattici** (parse trees) che definiscono la struttura sintattica degli elementi di L(G).

L'albero sintattico di un'assegnata sentenza in L(G) descrive l'insieme delle derivazioni che, applicate al simbolo di start, generano la sentenza. Per associare a ogni albero sintattico un'unica sequenza di derivazioni si assume la convenzione di procedere, in un albero sintattico parziale, a *espandere sempre il non terminale più a sinistra*.

Di seguito è mostrato l'albero sintattico che la grammatica delle espressioni consente di associare alla sentenza **b*b + a*c*d**.

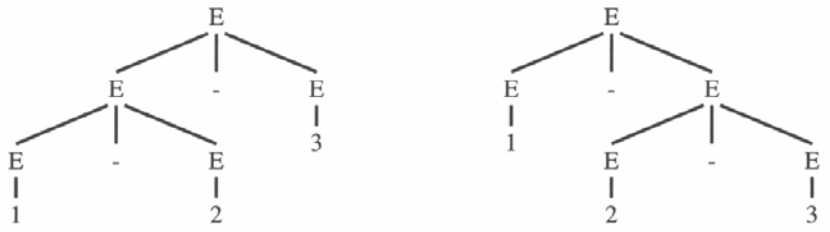


Una grammatica G si dice **ambigua** se esiste almeno una sentenza in L(G) per la quale è possibile determinare due o più diversi alberi sintattici[3].

Una semplice grammatica ambigua è la seguente:

E = **1** | **2** | **3** | E - E

Per rendersene conto, basta osservare che esistono due diversi alberi sintattici per l'espressione **1-2-3**, corrispondenti rispettivamente alle due diverse interpretazioni **(1-2)-3** e **1-(2-3)** dell'espressione fornita. Peraltro, a queste due diverse interpretazioni corrispondono due diversi risultati numerici.



Pertanto, in generale, all'ambiguità della grammatica corrisponde l'ambiguità d'interpretazione di alcune delle sentenze in L(G) [4].

Purtroppo, il problema dell'ambiguità di una grammatica libera da contesto G è *indecidibile*, ossia non esiste un algoritmo che, data una grammatica G, sia in grado di stabilire se G è ambigua oppure no. Parimenti è indecidibile il problema dell'intrinseca ambiguità di un linguaggio. Ad ogni modo, fortunatamente, è molto spesso possibile eliminare l'ambiguità di una grammatica, trasformandola in una grammatica non ambigua.

[3] Esistono linguaggi liberi da contesto **intrinsecamente ambigui**, ossia tali che tutte le grammatiche che li generano sono ambigue.
[4] Questo va assolutamente evitato nella costruzione di interpreti e compilatori.

LINGUAGGI, GRAMMATICHE ED ESPRESSIONI REGOLARI

Linguaggi e grammatiche regolari

Le **grammatiche regolari** costituiscono un'utile sottoclasse delle grammatiche libere da contesto. Esse possono essere definite come il sottoinsieme delle grammatiche libere da contesto che impiegano la ricorsività solo per esprimere la ripetizione. Se si fa riferimento alla EBNF, si definiscono regolari le grammatiche libere da contesto che non impiegano la ricorsività.

I linguaggi definibili usando grammatiche regolari prendono il nome di **linguaggi regolari** [5].

Un esempio di grammatica regolare è:

Identificatore = Lettera {Lettera | Cifra}
Lettera = **a** | … | **z** | **A** | … | **Z**
Cifra = **0** | **1** | … | **9** [6]

La grammatica definisce il linguaggio regolare degl'identificatori, stringhe di lettere e cifre inizianti con una lettera.

Espressioni regolari

I linguaggi regolari possono essere rappresentati, oltre che dalle grammatiche regolari, utilizzando una *notazione algebrica*, quella delle **espressioni regolari**.

Dato l'insieme di simboli terminali T, sussistono le seguenti regole di formazione delle espressioni regolari:

1. **ε** è un'espressione regolare;
2. ogni simbolo **x** ∈ **T** è un'espressione regolare;
3. se **R** è un'espressione regolare, l'*espressione parentesizzata* **(R)** è un'espressione regolare;
4. se **R1** e **R2** sono espressioni regolari, la *concatenazione* di **R1** e **R2**, scritta **R1 R2**, è un'espressione regolare;
5. se **R1** e **R2** sono espressioni regolari, l'*alternativa*, scritta **R1 | R2** oppure **R1 + R2**, è un'espressione regolare;
6. se **R** è un'espressione regolare, la *ripetizione* di **0** o più occorrenze di **R**, scritta **R*** dove ***** è l'operatore di *chiusura di Kleene*, è un'espressione regolare;
7. nient'altro è un'espressione regolare.

Allo scopo di ridurre l'uso delle parentesi, è anche definito un ordine di precedenza tra i tre operatori: la chiusura ha la massima priorità, segue la concatenazione e, per ultima, l'alternativa.

Il valore di un'espressione regolare su T è un linguaggio definito su T, ossia un sottoinsieme di T*. Pertanto, se **R** è un'espressione regolare, il suo valore si denota con **L(R)** e si ottiene applicando all'espressione regolare le seguenti regole di valutazione:

1. L(**x**) = { **x** }, se **x** ∈ **T** ∪ { **ε** };
2. L(**(R)**) = L(**R**);
3. L(**R1 R2**) = L(**R1**)◦L(**R2**), ossia è il linguaggio che si ottiene concatenando a ciascuna sentenza di L(**R1**) tutte le sentenze di L(**R2**);
4. L(**R1 | R2**) = L(**R1**) ∪ L(**R2**), ossia è il linguaggio che si ottiene unendo i linguaggi L(**R1**) e L(**R2**);
5. L(**R***) = L(**R**)* = L(**ε**) ∪ L(**R**) ∪ L(**R R**) ∪ L(**R R R**) ∪ …, ossia l'unione della stringa vuota, di tutte le stringhe in L(**R**), di tutte le stringhe formate da coppie, triple, ecc. di stringhe in L(**R**).

Ad esempio, l'espressione regolare **(0|1)*11(1|01)*(ε|0)** denota il linguaggio formato dalle stringhe di

[5] Secondo la classificazione di Chomsky, i linguaggi regolari appartengono al livello 3.
[6] I tre puntini … sono usati solo per comodità, con l'ovvio significato, ma non sono metasimboli legali.

qualsiasi lunghezza di **0** e **1** [**(0|1)***] che terminano con due **1** [**11**] e sono seguite da stringhe di **0** e **1** di qualsiasi lunghezza in cui ogni **0** è immediatamente seguito da un **1** [**(1|01)***], eventualmente terminate con uno **0** [**(ε|0)**].

Linguaggi regolari ed espressioni regolari sono legati dalla seguente proprietà: tutti i linguaggi definiti da espressioni regolari sono regolari, mentre tutti i linguaggi regolari sono valori di espressioni regolari.

In altri termini, dato un linguaggio regolare **L** esiste almeno un'espressione regolare **r** il cui valore è **L**; viceversa, per ogni espressione regolare **r** avente valore **L** esiste almeno una grammatica regolare **G** tale che **L(G) = L**.

Linguaggi regolari e automi a stati finiti

L'utilità dei linguaggi regolari sta nel fatto che il riconoscimento di sentenze regolari, ossia la verifica che una data stringa di simboli terminali appartiene a un linguaggio regolare, può essere realizzato da un opportuno **automa a stati finiti** deterministico (ASFD), agevolmente ed efficientemente simulabile da un programma.

Più precisamente, dato un qualsiasi linguaggio regolare L, è sempre possibile determinare un ASFD che accetta le sole sentenze di L. Viceversa, dato un qualsiasi ASFD, l'insieme delle stringhe che questo accetta è sempre un linguaggio regolare.

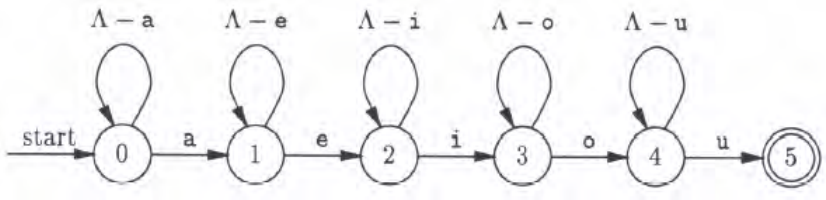
In realtà, la determinazione di un automa a stati finiti in grado di riconoscere o accettare le sole sentenze di un linguaggio regolare è più agevole se si rilascia il vincolo che l'automa debba essere deterministico e si procede quindi ad individuare un automa non deterministico (ASFND).

L'automa non deterministico può poi essere **sempre** trasformato in un automa deterministico equivalente, ossia in grado di riconoscere lo stesso linguaggio.

Una rappresentazione grafica molto utilizzata di un automa è il suo **diagramma degli stati**, in cui l'automa è rappresentato mediante un grafo orientato: i nodi rappresentano gli stati, mentre gli archi orientati rappresentano le transizioni; quindi, ogni arco è etichettato con i simboli d'ingresso la cui lettura determina la transizione. Gli stati individuati da nodi con un doppio cerchio sono gli **stati finali**. Lo stato individuato tramite una freccia entrante è detto **stato iniziale**.

L'ipotesi di determinismo di un automa implica che, per ognuno dei suoi stati e per ognuno dei suoi simboli d'ingresso, esiste al più una transizione uscente dal nodo associato allo stato etichettata con il simbolo.

La computazione di un ASFD ha inizio dallo stato iniziale e sviluppa una sequenza di transizioni. Per ogni successivo simbolo della stringa d'ingresso, l'automa verifica se, a partire dallo stato corrente, esiste una transizione etichettata con il simbolo. In tal caso, la transizione è realizzata, con la modifica dello stato corrente. La computazione termina dopo avere letto l'ultimo simbolo della stringa d'ingresso, ovvero quando per il simbolo letto non è definita alcuna transizione. Se, al termine della computazione, l'automa è in uno stato finale, la stringa è accettata, ossia riconosciuta come una sentenza del linguaggio regolare; altrimenti, la stringa è rigettata come non appartenente al linguaggio.

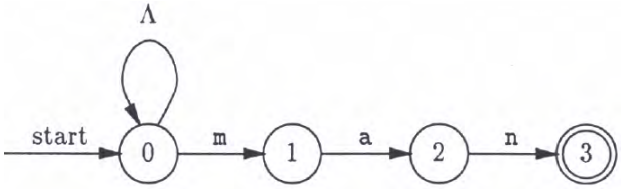


Ad esempio, l'ASFD riportato in figura riconosce le stringhe di lettere da cui è possibile estrarre, tramite cancellazione dei caratteri superflui, la sottostringa **aeiou**. **Λ** denota l'insieme delle lettere minuscole e maiuscole, mentre **Λ-a** denota l'insieme di tutte le lettere diverse da **a**. Raggiunto lo stato finale **5**, la stringa è accettata, anche se non è stata completamente letta. Se il carattere letto non è una lettera, la lettura si arresta e la stringa è rigettata.

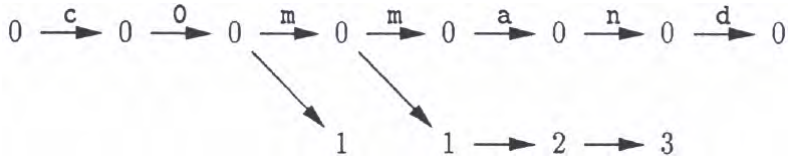
In un automa non deterministico è possibile che, per qualcuno dei suoi stati e per qualcuno dei suoi simboli d'ingresso, esista più di una transizione uscente dal nodo associato allo stato etichettata con il simbolo.

Dalla sua stessa definizione deriva che un ASFND sviluppa, per una data stringa d'ingresso, non una ma più computazioni. Alternativamente si può dire che l'automa sviluppa una sola *computazione non deterministica* che si caratterizza, per ogni simbolo letto, da una transizione che trasforma l'insieme degli stati correnti in un nuovo insieme di stati. Al termine della computazione, la stringa è accettata solo se l'insieme degli stati in cui l'automa si trova contiene almeno uno stato finale.

Ad esempio, l'ASFND riportato in figura riconosce tutte le stringhe di lettere che contengono la sotto-stringa **man**.



Il comportamento dell'automa quando legge la stringa **command** è mostrato nella figura seguente:

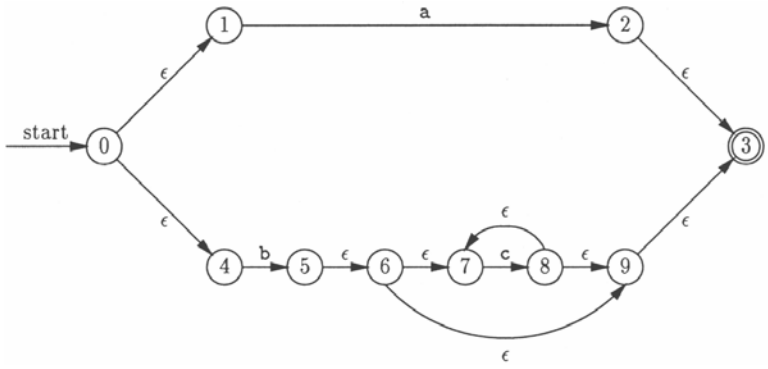


In risposta agl'ingressi **c** e **o** l'automa rimane nello stato **0**. Quando legge la prima **m**, l'automa ha due alternative: rimanere in **0** o andare in **1**. Quando legge la seconda **m** non può effettuare alcuna transizione a partire dallo stato **1**, mentre ha ancora due alternative a partire da **0**. A partire dallo stato **0**, la lettura di **a**, **n** e **d** produce il permanere in **0**. Invece, a partire dallo stato **1**, la lettura di **a** e **n** produce la transizione prima a **2** e poi a **3**; la successiva lettura di **d** non produce alcuna ulteriore transizione. L'insieme degli stati in cui l'automa può trovarsi al termine della computazione è quindi {**0**, **3**}. Poiché di questi, **3** è uno stato finale, la stringa **command** è riconosciuta contenere **man**.

Automi non deterministici con ε-transizioni

Per i motivi che saranno chiari nel seguito, introduciamo ora gli ASFND dotati di ε-transizioni. In questi automi esistono degli archi orientati etichettati con il simbolo **ε**, che corrispondono a transizioni che, durante una computazione non deterministica, possono essere percorsi senza che venga "consumato" alcun simbolo della stringa d'ingresso.

La figura mostra un ASFND con ε-transizioni in grado di riconoscere tutte e sole le stringhe del linguaggio regolare definito dall'espressione **a|bc***.



Ad esempio, la computazione che porta all'accettazione della stringa **bcc**, appartenente al linguaggio, si snoda attraverso la sequenza di stati e di transizioni seguente:

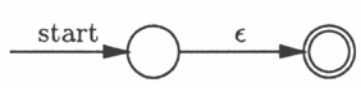
$$0 \rightarrow \varepsilon \rightarrow 4 \rightarrow b \rightarrow 5 \rightarrow \varepsilon \rightarrow 6 \rightarrow \varepsilon \rightarrow 7 \rightarrow c \rightarrow 8 \rightarrow \varepsilon \rightarrow 7 \rightarrow c \rightarrow 8 \rightarrow \varepsilon \rightarrow 9 \rightarrow \varepsilon \rightarrow 3$$

L'ASFND che riconosce un linguaggio regolare

L'introduzione degli ASFND dotati di ε-transizioni rende estremamente agevole convertire un'espressione regolare nell'automa in grado di accettarne il linguaggio, usando un algoritmo che va sotto il nome di *costruzione di Thompson*.

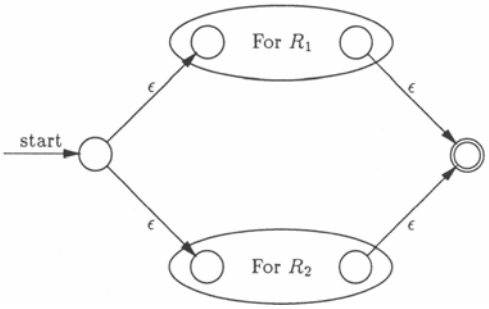
Per descrivere l'algoritmo, facciamo riferimento all'espressioni regolari costruibili sull'insieme di simboli terminali T e, se **r** è un'espressione regolare su T denotiamo con ASFND(**r**) l'automa che accetta il linguaggio definito da **r**. Ciò posto, la costruzione di Thompson specifica per ognuna delle regole 1-6 di formazione di un'espressione regolare il corrispondente automa:

1. **ASFND(ε)** è

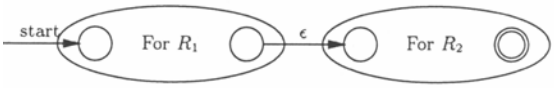


2. ∀ **x** ∈ **T**, **ASFND(x)** è

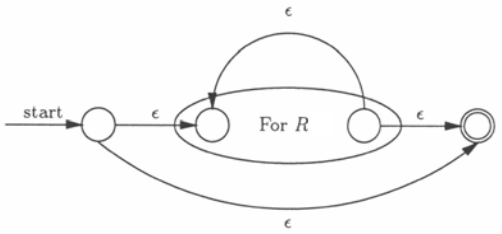
3. Se **R** è un'espressione regolare, ASFND(**(R)**) = ASFND(**R**)
4. Se $\mathbf{R_1}$ e $\mathbf{R_2}$ sono espressioni regolari, ASFND($\mathbf{R_1 \mid R_2}$) è



5. Se $\mathbf{R_1}$ e $\mathbf{R_2}$ sono espressioni regolari, ASFND($\mathbf{R_1\ R_2}$) è



6. Se **R** è un'espressione regolare, ASFND(**R***) è



Nei diagrammi mostrati, ciascun ovale sintetizza un intero automa di cui si mostrano solo lo stato di start e quello finale.

È agevole verificare che l'applicazione delle regole precedenti all'espressione **a|bc*** consente di costruire l'automa mostrato nella pagina precedente.

Gli automi che la costruzione di Thompson consente di ottenere hanno le seguenti proprietà:

1. Sono dotati di uno stato di start e uno stato di accettazione, quest'ultimo privo di archi uscenti.
2. Hanno nodi con un solo arco uscente, etichettato con uno dei simboli di T, oppure con al più due archi uscenti etichettati con ε.
3. Hanno un numero di nodi pari al più al doppio del numero complessivo di simboli e di operatori presenti nell'espressione regolare.

L’eliminazione delle ε-transizioni

Gli automi con ε-transizioni possono sempre essere trasformati in automi senza ε-transizioni, utilizzando l’algoritmo descritto di seguito.

Prima di tutto, in un automa con ε-transizioni si definiscono *stati importanti* lo stato di start e i soli stati dotati di transizioni entranti etichettate con simboli di T.

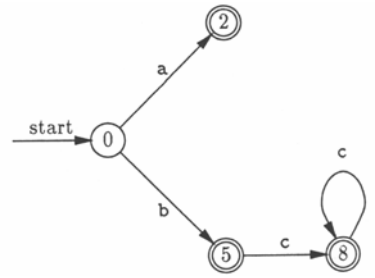
Ad esempio, nell’automa mostrato a pag. 6, sono stati importanti esclusivamente: **0**, che è lo stato di start; **2**, che ha una transizione entrante etichettata con **a**; **5**, che ha una transizione entrante etichettata con **b**; **8**, che ha una transizione entrante etichettata con **c**.

Dato un ASFND con ε-transizioni, l’automa equivalente privo di ε-transizioni conserva i soli stati importanti dell’automa originario. Inoltre, se *i* e *j* sono due stati importanti, il nuovo automa ha una transizione che porta da *i* a *j* etichettata con il simbolo **x** se esiste uno stato *k* dell’automa originario tale che:

1. Nell’automa originario, vi è una transizione, etichettata con **x**, che porta da *k* a *j*.
2. Lo stato *k* è raggiungibile dallo stato *i* lungo un cammino di 0 o più ε-transizioni.

Inoltre, lo stato importante *i* è uno stato finale del nuovo automa se, nell’automa originario esiste un cammino di 0 o più ε-transizioni che porta allo stato finale.

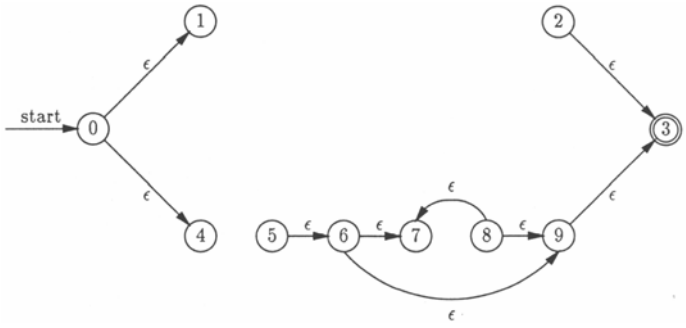
Applicando questo algoritmo all’automa di pag. 6 si ottiene:



Il nuovo automa ha i soli stati importanti **0**, **2**, **5** e **8**, di cui **2**, **5** e **8** finali. Inoltre, in esso ci sono:

1. Una transizione **a** che porta da **0** a **2**; infatti nell’automa originario c’è una transizione **a** che porta da **1** a **2** e **1** è raggiungibile da **0** tramite sole ε-transizioni.
2. Una transizione **b** che porta da **0** a **5**; infatti nell’automa originario c’è una transizione **b** che porta da **4** a **5** e **4** è raggiungibile da **0** tramite sole ε-transizioni.
3. Una transizione **c** che porta da **5** a **8**; infatti nell’automa originario c’è una transizione **c** che porta da **7** a **8** e **7** è raggiungibile da **5** tramite sole ε-transizioni.
4. Una transizione **c** che porta da **8** a **8**; infatti nell’automa originario c’è una transizione **c** che porta da **7** a **8** e **7** è raggiungibile da **8** tramite sole ε-transizioni.

La determinazione delle transizioni presenti nel nuovo automa è facilitata se si costruisce un nuovo automa, con gli stessi nodi di quello originario e le sole ε-transizioni, e lo si utilizza per costruire la *tabella di raggiungibilità* che specifica, per ogni coppia di nodi, se questi sono collegati tramite un cammino di 0 o più ε-transizioni.



	0	1	2	3	4	5	6	7	8	9
0	1	1			1					
1		1								
2			1	1						
3				1						
4					1					
5				1		1	1	1		1
6				1			1	1		1
7								1		
8				1				1	1	1
9				1						1

La trasformazione di un ASFND in un ASFD

Nell'esempio precedente, la semplice eliminazione delle ε-transizioni trasforma l'automa in deterministico. Ma questo non è vero sempre e, in generale, la trasformazione dell'automa in deterministico richiede l'applicazione di un apposito algoritmo.

L'idea generale che è alla base della trasformazione dell'automa da non deterministico a deterministico è la seguente: ogni stato dell'ASFD corrisponde ad un sottoinsieme degli stati dell'ASFND. Pertanto, se *n* è il numero degli stati dell'ASFND è possibile che gli stati dell'ASFD raggiungano il numero limite di 2^n. Ma per fortuna questo accade raramente, in pratica.

L'algoritmo di trasformazione dell'automa da non deterministico a deterministico è:

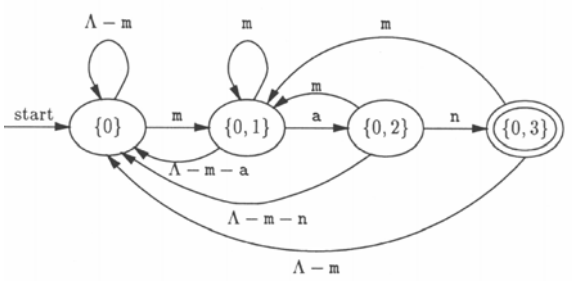
```
Poni ε-closure(0) come unico stato (di start), non marcato, in ASFD
while (∃ uno stato non marcato in ASFD)
  Scegli uno stato S non marcato in ASFD e marcalo
                                  // S è un insieme di stati di ASFND
  ∀ x ∈ T
    Z = ε-closure(move(S, x))
    if (Z non è uno stato di ASFD)
      Aggiungi Z, non marcato, agli stati di ASFD
      Rendi Z stato finale di ASFD se contiene almeno
           uno stato finale di ASFND
    Aggiungi ad ASFD una transizione etichettata x che va da S a Z
```

In esso, con riferimento all'ASFND:

- *0* denota lo stato di start.
- ε-closure(*i*) denota l'insieme degli stati raggiungibili a partire dallo stato *i* seguendo cammini di 0 o più ε-transizioni. Nella tabella di raggiungibilità, ε-closure(*i*) è l'insieme degli stati le cui colonne hanno un 1 nella riga associata allo stato *i*.
- ε-closure(*S*), dove *S* è un insieme di stati, denota l'unione delle ε-closure(*i*), $\forall\ i \in S$.
- move(*S*, **x**), dove *S* è un insieme di stati e **x** un simbolo, denota l'insieme degli stati raggiungibili da qualche stato di *S* per effetto di una transizione etichettata **x**.

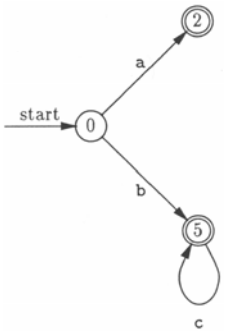
L'algoritmo può essere utilizzato sia a partire dall'automa con ε-transizioni che da quello privo di ε-transizioni. L'unica differenza sta che, in questo secondo caso, ε-closure(*i*) denota l'insieme degli stati direttamente raggiungibili a partire dallo stato *i*.

Applicando l'algoritmo mostrato all'ASFND che riconosce tutte le stringhe di lettere che contengono la sottostringa **man**, si ottiene l'ASFD equivalente mostrato.



La minimizzazione del numero degli stati di un ASFD

Dato un ASFD è spesso possibile trovarne un altro ad esso equivalente con un numero minore di stati interni e, quindi, in qualche senso, più semplice. Ad esempio, per l'automa di pag. 8 è agevole verificare che gli stati **5** e **8** sono indistinguibili ed è possibile quindi fonderli.



Gli algoritmi di minimizzazione del numero degli stati interni qui utilizzabili sono quelli studiati nell'insegnamento di Reti logiche. Gli automi d'interesse in questo ambito possono essere assimilati ad automi di Moore con uscita binaria pari a 1 per gli stati finali e a 0 per tutti gli altri.

IL PARSING DEI LINGUAGGI LIBERI DA CONTESTO

Linguaggi liberi da contesto e automi a pila

Il problema dell'accettazione di sentenze libere da contesto, ossia della verifica che una data stringa di simboli terminali appartiene a un linguaggio libero da contesto trova soluzione generale grazie all'utilizzo di **automi a pila** (o stack).

Gli automi a pila sono degli automi a stati finiti resi più potenti dalla disponibilità di una memoria aggiuntiva a pila (stack)[7]. Ad ogni passo, questi automi, in base a stato corrente, prossimo simbolo terminale e simbolo presente al top dello stack, effettuano una transizione di stato ed aggiornano lo stack.

Il processo realizzato dall'automa a pila prende il nome di **parsing**. Sostanzialmente esso equivale al tentativo di costruzione dell'albero sintattico (unico se la grammatica non è ambigua) associato alla stringa di simboli terminali. Se la costruzione ha successo, la stringa è accettata come sentenza del linguaggio L(G), altrimenti è rigettata come non appartenente al linguaggio.

Dato un qualsiasi linguaggio libero da contesto L, è sempre possibile determinare un automa a pila non deterministico (APND) che accetta le sole sentenze di L. Viceversa, dato un qualsiasi APND, l'insieme delle stringhe che questo accetta è sempre un linguaggio libero da contesto.

A differenza di ciò che accade con i linguaggi regolari, dove il non determinismo degli automi a stati finiti non introduce maggiore potere computazionale, nel caso dei linguaggi liberi da contesto, il non determinismo degli automi a pila gioca un ruolo fondamentale. Infatti, gli automi a pila deterministici (APD) sono in grado di riconoscere solo un sottoinsieme proprio dei linguaggi liberi da contesto[8].

Gli automi a pila deterministici sono comunque di grande rilevanza. Pur non essendo sempre utilizzabili, sono in grado di operare con molte grammatiche libere da contesto, effettuando il parsing con complessità temporale O(n), ossia lineare nella lunghezza n della stringa d'accettare[9].

Il parsing discendente

Il **parsing discendente** (top down parsing) tenta di costruire l'albero sintattico associato alla stringa da accettare a partire dalla radice e creando i nodi dell'albero in ordine anticipato.

In generale, il parsing discendente può richiedere **backtracking**, ossia l'annullamento di alcune scelte già effettuate e la scansione ripetuta di parti della stringa d'accettare.

Ad esempio, consideriamo la grammatica:

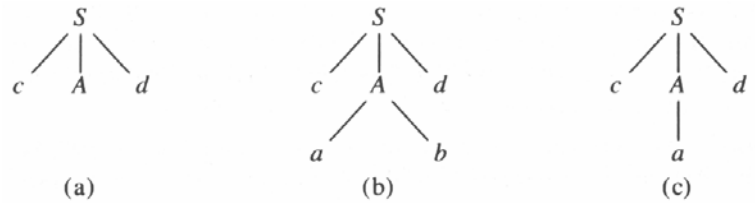
S = **c** A **d**
A = **a b** | **a**

e la stringa di simboli terminali **cad** e proviamo a costruire il relativo albero sintattico con modalità discendente. Inizialmente l'albero consta del solo nodo radice, associato al simbolo di start S, e **c** è il carattere corrente della stringa. Poiché esiste una sola produzione che ha S come parte sinistra, espandiamo l'albero, ottenendo quello mostrato in (a). Poiché la foglia più a sinistra è il terminale **c** e coincide quindi con il carattere corrente, spostiamoci sulla foglia alla destra, facendo nel contempo diventare corrente il carattere successivo della stringa (**a**). La foglia successiva nell'albero (a) è il non terminale A per il quale esistono nella grammatica due distinte produzioni. Proviamo l'espansione di A con la prima delle due produzioni, ottenendo l'albero (b). Ora la foglia da considerare è il terminale **a**; coincidendo questo con il terminale corrente, spostiamoci sulla foglia successiva, facendo nel contempo diventare corrente il carattere successivo (**d**).

[7] Una pila è una memoria potenzialmente illimitata.

[8] Gli automi deterministici ottenuti rilassando i vincoli imposti dalla politica LIFO, ossia sostituendo la pila con un array in cui tutti gli elementi siano accessibili, sono in grado di accettare qualunque linguaggio libero da contesto.

[9] I principali metodi generali di parsing, utilizzabili per qualsiasi grammatica libera da contesto, sono l'algoritmo di Early e quello di Cocke-Younger-Kasami (CYK), che hanno complessità temporale O(n^3), ossia cubica nella lunghezza n della stringa d'accettare.



Questa volta, il confronto tra la foglia **b** e il carattere corrente **d** non ha successo. Siamo quindi in presenza di un fallimento che impone di attuare una fase di backtracking, ossia di ritornare all'ultima scelta effettuata annullandola, per seguire una diversa alternativa, se esiste. Dobbiamo quindi ritornare alla situazione precedente all'espansione del non terminale A, ossia all'albero (a) e al carattere corrente **a**. Questa volta, espandiamo il non terminale A con la seconda produzione, ottenendo l'albero (c). Ora il confronto tra la foglia da esaminare **a** e il carattere corrente ha successo. Possiamo quindi spostarci sulla foglia successiva, facendo nel contempo diventare corrente il carattere successivo (**d**). Anche questa volta il confronto ha successo e, poiché la stringa in fase di accettazione è terminata, possiamo concludere che essa è una sentenza del linguaggio e che il suo albero sintattico è l'albero (c).

La ricorsività a sinistra di una grammatica e la sua eliminazione

Il parsing discendente non è realizzabile se la grammatica è *ricorsiva a sinistra*. Infatti, ad esempio, la presenza nella grammatica della produzione $A = A\,\beta$ implica che l'espansione di A produce nuovamente l'esigenza di espandere A e così via, all'infinito.

Per caratterizzare esattamente la **ricorsività a sinistra**, introduciamo la notazione $\alpha \Rightarrow^{+} \beta$ per indicare la derivazione in uno o più passi di β da α, ossia l'esistenza di una sequenza di una o più produzioni che, applicate ordinatamente alla stringa α e a quelle man mano ottenute, generano la stringa β; le stringhe α e β possono contenere sia terminali che non terminali [10].

Ciò posto, diciamo che una grammatica è ricorsiva a sinistra se essa ha almeno un simbolo non terminale A per il quale esiste una derivazione $A \Rightarrow^{+} A\beta$, con β stringa qualsiasi. In particolare, la ricorsività a sinistra si dice **immediata** se la derivazione ha lunghezza unitaria, ossia se esiste una produzione della forma $A = A\,\beta$.

Per rendere possibile l'applicazione del parsing discendente ad una grammatica ricorsiva a sinistra, occorre trasformare la grammatica in modo che non presenti più la ricorsività a sinistra.

La ricorsività a sinistra immediata può essere eliminata usando l'algoritmo:

$\forall$ produzione immediatamente ricorsiva a sinistra
$A = A\beta_1 \mid A\beta_2 \mid \ldots \mid A\beta_n \mid \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m$
Sostituisci la produzione originaria con le due produzioni
$A = \alpha_1 N \mid \alpha_2 N \mid \ldots \mid \alpha_m N$
$N = \beta_1 N \mid \beta_2 N \mid \ldots \mid \beta_n N \mid \varepsilon$
con N simbolo non terminale non presente nella grammatica

Ad esempio, la sua applicazione alla grammatica:

E = E + T | T
T = T * F | F
F = (E) | **a** | **b** | **c** | **d**

la trasforma nella grammatica:

E = T E'
E' = + T E' | ε
T = F T'
T' = * F T' | ε
F = (E) | **a** | **b** | **c** | **d**

[10] $\alpha \Rightarrow^{*} \beta$ indica che β deriva in 0 o più passi da α; ovviamente la derivazione è in 0 passi se e solo se $\alpha = \beta$

Nel caso di ricorsività a sinistra indiretta si può adoperare l'algoritmo generale [11]:

```
Ordina in qualche modo i non terminali della grammatica A1, A2, …, Am
∀ i ≤ m
  ∀ j < i
    if (∃ produzione Ai = Ajβ, con Aj = α1 | α2 | … | αk)
      Sostituisci Ai = Ajβ con Ai = α1β | α2β | … | αkβ
  Elimina la ricorsività a sinistra immediata dalla produzione Ai = …
```

La sua applicazione alla grammatica [12]: S = A **a** | **b**
A = A **c** | S **d** | ε

certamente ricorsiva a sinistra, in quanto $S \Rightarrow A\ \mathbf{a} \Rightarrow S\ \mathbf{d}\ \mathbf{a}$, e quindi $S \Rightarrow^{+} S\ \mathbf{d}\ \mathbf{a}$, la trasforma nella grammatica (ottenuta con l'ordinamento S, A dei non terminali):

S = A **a** | **b**
A = **b d** A' | A'
A' = **c** A' | **a d** A' | ε

La fattorizzazione sinistra di una grammatica

L'eliminazione della ricorsività a sinistra dalla grammatica garantisce che il parsing discendente possa operare senza il rischio di entrare in un ciclo infinito. Durante la fase di parsing è però sempre possibile che si verifichi backtracking.

La **fattorizzazione sinistra** è una trasformazione della grammatica che ha lo scopo di ridurre il rischio di backtracking. L'idea è che, quando non è chiaro quale di due produzioni debba essere utilizzata per espandere un simbolo non terminale A, conviene riscrivere le produzioni di A in modo tale da rinviare la decisione al momento in cui si hanno maggiori informazioni sulla stringa in fase d'accettazione.

Ad esempio, se la grammatica ha le produzioni:

$$A = \alpha\ \beta_1 \mid \alpha\ \beta_2$$

la fattorizzazione sinistra le trasforma in:

$$A = \alpha\ A'$$
$$A' = \beta_1 \mid \beta_2$$

con A' non terminale non presente nella grammatica originaria. In questo modo, A può essere espanso senza ambiguità in α A', rinviando la scelta tra β_1 o β_2 ad un momento successivo, quando si avranno ulteriori informazioni sulla stringa in fase d'accettazione.

L'algoritmo generale di fattorizzazione sinistra è:

```
while (La grammatica ha alternative aventi un prefisso comune)
  ∀ non terminale A
    Trova, se esiste, il più lungo prefisso α ≠ ε comune a due o più sue
    alternative: A = αβ1 | αβ2 | … | αβn | γ
    dove γ sintetizza tutte le alternative che non cominciano con α
    e sostituisci la produzione precedente con la coppia
                A = αA' | γ e     A' = β1 | β2 | … | βn
```

La sua applicazione alla grammatica: S = **i** E **t** S | **i** E **t** S **e** S | **a**
E = **b**

produce la grammatica fattorizzata a sinistra:

[11] L'algoritmo può operare su qualsiasi grammatica che non abbia produzioni nulle (A = ε) o cicli (derivazioni della forma $A \Rightarrow^{+} A$). Comunque, produzioni nulle e cicli sono anch'essi eliminabili da ogni grammatica.
[12] Tecnicamente, l'algoritmo potrebbe non funzionare, in quanto la grammatica ha una ε-produzione; ma, in realtà la trasformazione è realizzata correttamente.

S = **i** E **t** S S'| **a**
S' = **e** S | ε
E = **b**

Le grammatiche LL(1)

Spesso, l'accurata scrittura della grammatica, l'eliminazione della ricorsività sinistra e la fattorizzazione sinistra consentono di ottenere una grammatica di cui può essere effettuato il **parsing discendente predittivo**, ossia tale da non richiedere il ricorso al backtracking.

Nel parsing discendente predittivo, noti il simbolo non terminale da espandere (il primo a partire da sinistra) e il simbolo corrente d'ingresso (il primo da sinistra ancora non considerato), è sempre possibile determinare quale debba essere la produzione da applicare.

Una grammatica che rende possibile il parsing discendente predittivo prende il nome di **grammatica LL(1)**, dove:

1. la prima **L** indica che la stringa di cui effettuare il parsing è analizzata da sinistra verso destra (**L**eft-to-right);
2. la seconda **L** indica che si procede sempre a derivare espandendo il simbolo non terminale più a sinistra (**L**eftmost-derivation);
3. **(1)** indica che è sufficiente la conoscenza del solo simbolo terminale corrente per rendere possibile la predizione.

Più in generale, si parla di grammatiche **LL(k)** quando è necessario analizzare *k* simboli terminali per potere effettuare la predizione ed evitare il backtracking.

Un linguaggio libero da contesto si dice **linguaggio LL(k)** se ammette una grammatica LL(k).

L'unione dei linguaggi LL(k) , per tutti i valori di k > 0, è un sottoinsieme proprio dei linguaggi liberi da contesto deterministici. In altri termini, esistono linguaggi liberi da contesto deterministici di cui non è possibile effettuare il parsing predittivo, ossia per i quali non esiste, per alcun valore di k > 0, una grammatica LL(k) [13].

Gl'insiemi FIRST e FOLLOW

Per un'assegnata grammatica G, se α è una stringa di simboli terminali e non terminali, **FIRST(α)** è l'insieme dei simboli terminali con cui possono iniziare le stringhe derivate da α. Se $\alpha \Rightarrow^* \varepsilon$, allora $\varepsilon \in$ FIRST(α).

Per un'assegnata grammatica G, se A è un simbolo non terminale, **FOLLOW(A)** è l'insieme dei terminali che possono apparire alla immediata destra di A in qualche forma sentenziale, cioè è l'insieme dei simboli terminali **a** per i quali esiste una derivazione $S \Rightarrow^* \alpha$ A **a** β, con α e β stringhe qualsiasi. Se A può essere il simbolo più a destra in qualche forma sentenziale, allora il simbolo **$** $\in$ FOLLOW(A), dove **$** è un simbolo non appartenente all'insieme dei terminali.

Ad esempio, per la grammatica:

E = T E'
E' = **+** T E' | ε
T = F T'
T' = ***** F T' | ε
F = **(** E **)** | **a** | **b** | **c** | **d**

le definizioni precedenti portano ai seguenti insiemi:

FIRST(**a**) = { **a** } FIRST(**b**) = { **b** } FIRST(**c**) = { **c** }
FIRST(**d**) = { **d** } FIRST(**+**) = { **+** } FIRST(*****) = { ***** }
FIRST(**(**) = { **(** } FIRST(**)**) = { **)** }
FIRST(E) = FIRST(T) = FIRST(F) = { **(**, **a**, **b**, **c**, **d** }

[13] Dati un linguaggio libero da contesto L e un valore di k, il problema dell'esistenza di una grammatica LL(k) per il linguaggio L è indecidibile.

FIRST(E') = { +, ε } FIRST(T') = { *, ε }
FOLLOW(E) = FOLLOW(E') = {), $ }
FOLLOW(T) = FOLLOW(T') = { +,), $ }
FOLLOW(F) = { +, *,), $ }

Gl'insiemi FIRST e FOLLOW sono importanti al fine di caratterizzare le grammatiche LL(1). Infatti, si può dimostrare che una grammatica G = <T, N, S, P> è LL(1) se e solo se gode delle due seguenti proprietà:

1. Due differenti produzioni dello stesso simbolo non terminale A non derivano mai stringhe che iniziano con lo stesso simbolo terminale; ossia se $A = \alpha \mid \beta$, allora $FIRST(\alpha) \cap FIRST(\beta) = \phi$.
2. Un simbolo non terminale A che può derivare ε non deriva mai stringhe che iniziano con uno dei simboli non terminali che possono seguire A; ossia se $A \Rightarrow^{+} \varepsilon$, allora $FIRST(A) \cap FOLLOW(A) = \phi$.

Qui di seguito, riportiamo gli algoritmi che consentono di costruire gl'insiemi FIRST e FOLLOW.

Il primo algoritmo consente di costruire l'insieme FIRST per ogni simbolo terminale e non terminale:

```
∀ x ∈ T, FIRST(x) = {x}
∀ A ∈ N, FIRST(A) = { }
∀ produzione A = ε, FIRST(A) = {ε}
do
   ∀ produzione A = A1 A2 … Ak
     ∀ i ≤ k
       FIRST(A) ∪ = (FIRST(Ai) – {ε})
       if ε ∉ FIRST(Ai) continue    // passa alla produzione seguente
       FIRST(A) ∪ = {ε}
while (Qualche insieme è stato modificato)
```

A partire dalla conoscenza degl'insiemi FIRST per ogni simbolo, è possibile determinare l'insieme FIRST di una stringa di simboli usando l'algoritmo:

```
FIRST(X1 X2 … Xn) = { }        // X1 X2 … Xn sono i simboli terminali e non
                               // terminali della stringa
∀ i ≤ n
   FIRST(X1 X2 ... Xn) ∪ = (FIRST(Xi) – {ε})
   if ε ∉ FIRST(Xi) exit       // termina il calcolo
FIRST(X1 X2 ... Xn) ∪ = {ε}
```

Infine, il seguente algoritmo consente di determinare l'insieme FOLLOW per ogni simbolo non terminale:

```
∀ A ∈ N, FOLLOW(A) = { }
FOLLOW(S) := {$}        // S è il simbolo di start
do
   ∀ produzione A = α B β
     FOLLOW(B) ∪ = (FIRST(β) – {ε})       // FIRST(ε) = {ε}
     if (β == ε || ε ∈ FIRST(β)) FOLLOW(B) ∪ = FOLLOW(A)
while (Qualche insieme è stato modificato)
```

È utile notare che:

1. FIRST e FOLLOW sono insiemi di terminali (più $ per FOLLOW ed eventualmente ε per FIRST). I non terminali non entrano mai a far parte degl'insiemi FIRST e FOLLOW.
2. Per calcolare FIRST(A) bisogna considerare le produzioni che hanno A alla sinistra.
3. Per calcolare FOLLOW(A) bisogna considerare le produzioni che hanno A alla destra.

Per la grammatica:

S = B c | D B
B = a b | c S
D = d | **ε**

in cui le lettere maiuscole denotano non terminali e le minuscole terminali, l'applicazione degli algoritmi precedenti produce:

X	FIRST(X)	FOLLOW(X)
D	{ d, **ε** }	{ a, c }
B	{ a, c }	{ c, $}
S	{ a, c, d }	{ $, c }
Bc	{ a, c }	
DB	{ d, a, c }	
ab	{ a }	
cS	{ c }	
D	{ d }	
ε	{ **ε** }	

La tabella di parsing

La **tabella di parsing** di una grammatica G è una matrice **M** che ha tante righe quanti sono i non terminali di G e tante colonne quanti sono i terminali di G, più una colonna per il simbolo **$**. La tabella di parsing è alla base della costruzione dell'APD in grado di effettuare il parsing dei linguaggi definiti da grammatiche LL(1) in un tempo lineare nella lunghezza della stringa. L'algoritmo seguente, basato sulla disponibilità degl'insiemi FIRST e FOLLOW costruisce la tabella per un'assegnata grammatica:

```
∀ produzione A = α, esegui le operazioni seguenti:
  ∀ terminale a ∈ FIRST(α),
    Inserisci la stringa α nella casella M[A, a]
    Se ε ∈ FIRST(α)
         ∀ b ∈ FOLLOW(A), incluso eventualmente $
           Inserisci la stringa α nella casella M[A, b]
```

L'algoritmo può essere applicato a una qualsiasi grammatica. Però, *solo se la grammatica è LL(1) si ottiene una tabella di parsing priva di caselle con più di una stringa*. Questa proprietà è spesso addirittura assunta come definizione di grammatica LL(1).

L'applicazione dell'algoritmo alla grammatica precedente consente di costruire la tabella di parsing, dalla quale risulta evidente che la grammatica non è LL(1):

	A	**B**	**C**	**D**	**$**
S	**B c** **D B**		**B c** **D B**	**D B**	
B					
D	**ε**		**ε**		

Come ulteriore esempio, consideriamo il linguaggio delle parentesi tonde e quadre bilanciate:

E = **(** E **)** | **[** E **]** | **ε**

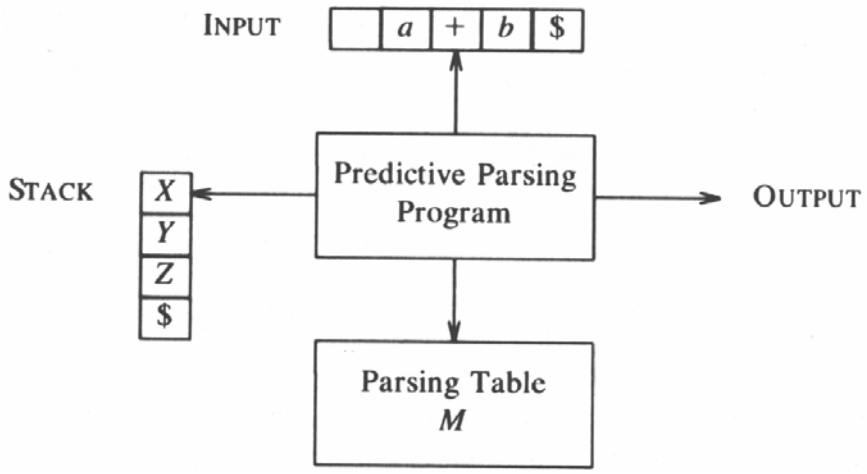
Applicando gli algoritmi precedenti si ottengono gl'insiemi FIRST e FOLLOW e la tabella di parsing:

X	FIRST(X)	FOLLOW(X)
E	{ (, [, **ε** }	{),], $ }
(E)	{ (}	
[E]	{ [}	
ε	{ **ε** }	

	(	)	[	]	$
E	(E)	ε	[E]	ε	ε

Il parser predittivo non ricorsivo

La disponibilità della tabella di parsing consente l'agevole costruzione di un **parser predittivo non ricorsivo**, i cui componenti sono mostrati in figura. Essi sono il buffer d'ingresso che contiene la stringa **w** in fase di accettazione terminata con il simbolo **$**, il cursore che specifica la posizione nel buffer d'ingresso del simbolo terminale corrente, uno stack di simboli, la tabella di parsing e lo stream di uscita, destinato a contenere, se **w** è una sentenza del linguaggio, la sua derivazione leftmost, un'indicazione di errore altrimenti.



L'algoritmo che controlla il funzionamento del parser è il seguente:

```
ip = puntatore al primo simbolo di w$
push($); push(S)        // stack inizialmente vuoto; S simbolo di start
do
   top = pop()
   if (top ∈ T || top == $)
      if (top != *ip) error()
      ip++
   else /* top ∈ N */
      if (M[top, *ip] è vuota) error();
      push(X1 X2 … Xn)          // con M[top, *ip] ≡ X1X2…Xn e X1 al top
      output(top = X1X2…Xn)
while (top != $)
```

Un esempio di esecuzione dell'algoritmo, con riferimento al linguaggio delle parentesi bilanciate e alla sentenza **([ ]) $** è:

```
input scandito        stack         azione
--------------        -----         ----------------------------
(                       E$          pop, push( (E) )
(                      (E)$         pop, (match), scan
([                      E)$         pop, push( [E] )
([                     [E])$        pop, (match), scan
([]                     E])$        pop, push( ε ) (no push)
([]                      ])$        pop, (match), scan
([])                      )$        pop, (match), scan
([])$                      $        pop, (match), scan
([])$                               stack vuoto: input accettato!
```

Con riferimento allo stesso linguaggio, il parsing della stringa **) $** produce:

```
input scandito        stack         azione
--------------        -----         ----------------------
)                        E$         pop, push( ε ) (no push)
)                         $         pop, (no match), errore!
```

Il parser predittivo ricorsivo

Dati un linguaggio e una grammatica LL(1) che lo descrive, è anche possibile costruire agevolmente un **parser predittivo ricorsivo** (o a discesa ricorsiva, in inglese recursive descent parser), utilizzando un qualunque linguaggio di programmazione in cui sia possibile scrivere funzioni ricorsive.

Se la grammatica LL(1) ha N simboli non terminali, il parser a discesa ricorsiva consta di N distinte funzioni, una per simbolo non terminale. Il corpo di ognuna delle funzioni è derivabile direttamente dalla parte destra della produzione che descrive nella grammatica il simbolo stesso.

Gli alberi sintattici astratti

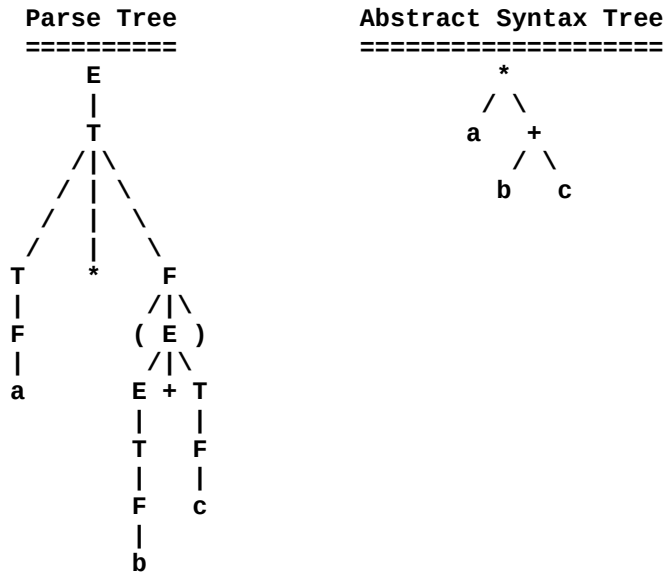
In un compilatore, il parser non può limitarsi ad effettuare il riconoscimento della stringa ricevuta, ma deve generare un output utilizzabile nelle fasi successive del processo di traduzione. Ad esempio, il parser predittivo non ricorsivo mostrato, fornisce in uscita l'elenco ordinato delle produzioni che, applicate alla stringa ricevuta, ne realizzano il riconoscimento.

Più spesso, i parser producono un **albero sintattico astratto** (AST, abstract syntax tree) che può considerarsi una forma condensata di albero sintattico, in cui:

1. Gli operatori appaiono come nodi interni invece che come foglie.
2. Le catene di produzioni singole sono *collassate*.
3. Le liste (di dichiarazioni, istruzioni, argomenti, ecc.) sono *appiattite*.
4. I dettagli sintattici (parentesi, segni di punteggiatura, ecc.) sono omessi.

In generale, un AST è una struttura migliore dell'albero sintattico per le fasi successive della compilazione, proprio in quanto omette i dettagli inessenziali che hanno a che fare con il linguaggio sorgente e contiene solo le informazioni essenziali sulla struttura del programma.

Un esempio di albero sintattico e del relativo AST per l'espressione aritmetica **`a*(b+c)`**, derivati a partire dalla consueta sintassi, sono:



Si noti come nell'AST non sono necessarie le parentesi, in quanto la struttura dell'albero definisce come le sottoespressioni sono raggruppate.

Per costrutti diversi dalle espressioni, la struttura dell'AST può essere liberamente definita, pur nel rispetto dei punti 1-4 elencati.

Un formalismo conveniente per gli AST è quello delle liste del Lisp, che possono essere così definite:

1. Un *atomo* è un dato elementare.
2. Una *lista* è una sequenza di zero o più atomi o liste racchiusa tra parentesi.

Ad esempio, **`(A (B 3) (C) ( ( ) ) )`** è una lista di 4 elementi di cui il primo è l'atomo **`A`**, il secondo la lista dei due atomi **`B`** e **`3`**, il terzo la lista dell'unico atomo **`C`** e il quarto la lista che ha come unico elemento la lista vuota.
Usando le liste, l'AST precedente può scriversi **`(* a (+ b c) )`**, in cui la forma linearizzata dell'albero è ottenuta realizzandone una visita in ordine anticipato.

LA TRADUZIONE DIRETTA DA SINTASSI

Grammatiche e regole di traduzione

Quando non è sufficiente il solo riconoscimento della correttezza sintattica di una sentenza, ma si vuole affiancare a questo un'elaborazione in grado di produrre un risultato, è necessario arricchire la grammatica del linguaggio con un insieme di **regole di traduzione**, una per ogni produzione. La regola di traduzione di una produzione A = α descrive la traduzione del non terminale A in funzione delle traduzioni dei non terminali in α e dei valori dei terminali in α.

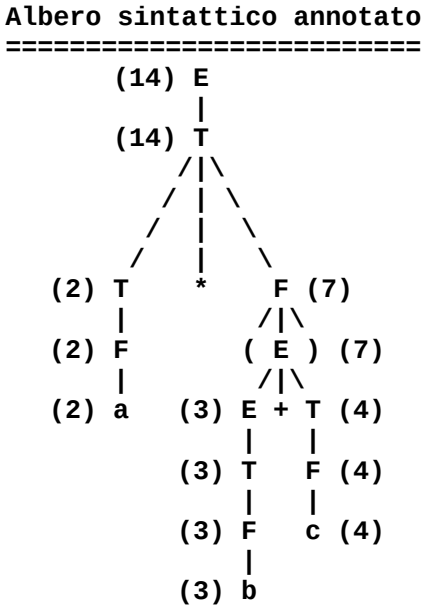
Ad esempio, la semplice grammatica delle espressioni già utilizzata può essere completata con le regole di traduzione mostrate:

Grammatica	Regole di traduzione
E = E + T	E_1.trans = E_2.trans + T.trans
E = T	E.trans = T.trans
T = T * F	T_1.trans = T_2.trans * F.trans
T = F	T.trans = F.trans
F = (E)	F.trans = E.trans
F = a	F.trans = a.value
F = b	F.trans = b.value
F = c	F.trans = c.value
F = d	F.trans = d.value

La notazione **`A.trans`** esprime il risultato della traduzione del non terminale **`A`**, mentre la notazione **`x.value`** denota il valore associato con il terminale **`x`**. Quando un non terminale è presente più di una volta, la regola di traduzione individua le diverse occorrenze per mezzo di pedici.

La più immediata traduzione di una sentenza del linguaggio ne richiede il preventivo riconoscimento, con la costruzione dell'albero sintattico, e la successiva applicazione delle regole sintattiche all'albero, procedendo bottom-up.

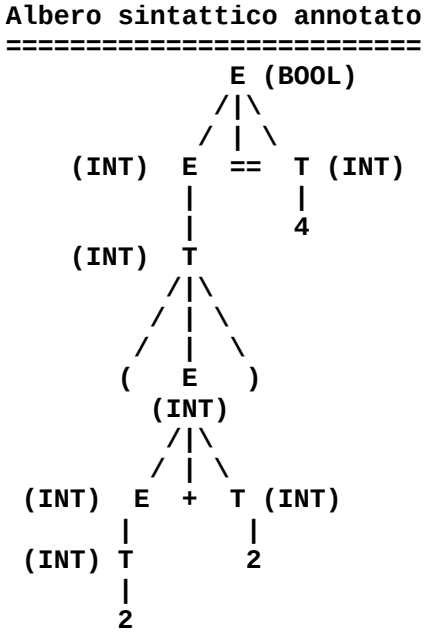
Ad esempio per l'albero sintattico dell'espressione aritmetica **`a * (b + c)`**, supponendo che i valori di **`a`**, **`b`** e **`c`** siano rispettivamente **`2`**, **`3`** e **`4`**, si ottiene l'albero annotato seguente:



L'uso delle regole di traduzione non è limitato all'interpretazione, ma può essere usato anche per realizzare il type checking di una espressione. L'esempio seguente mostra come:

Grammatica	Regole di traduzione
E = E + T	if ((E_2.trans==INT) && (T.trans==INT) E_1.trans = INT else E_1.trans = ERROR
E = E && T	if ((E_2.trans==BOOL) && (T.trans==BOOL) E_1.trans = BOOL else E_1.trans = ERROR
E = E == T	if ((E_2.trans==T.trans) && (E_2.trans!=ERROR)) E_1.trans = BOOL else E_1.trans = ERROR
E = T	E.trans = T.trans
T = true	T.trans = BOOL
T = false	T.trans = BOOL
T = int	T.trans = INT
T = (E)	T.trans = E.trans

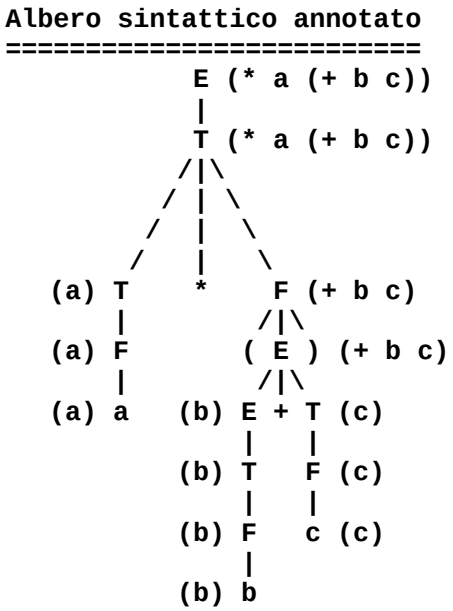
L'applicazione delle regole all'albero sintattico della sentenza **(2 + 2) == 4** produce l'albero annotato seguente, dal quale si evince che le regole di tipo sono rispettate e che il risultato dell'espressione è di tipo booleano.



Le regole sintattiche possono essere usate anche per generare l'albero sintattico astratto.

Grammatica	Regole di traduzione
E = E + T	E_1.trans = list(+, E_2.trans, T.trans)
E = T	E.trans = T.trans
T = T * F	T_1.trans = list(*, T_2.trans, F.trans)
T = F	T.trans = F.trans
F = (E)	F.trans = E.trans
F = a	F.trans = a
F = b	F.trans = b
F = c	F.trans = c
F = d	F.trans = d

dove **`list`** denota la primitiva di costruzione di una lista. L'applicazione di queste regole all'albero sintattico dell'espressione **`a * (b + c)`** determina l'albero sintattico annotato seguente:



e quindi costruisce la lista **`(* a (+ b c))`**, versione linearizzata dell'AST.

Il parsing discendente e la traduzione diretta da sintassi

Anche se è possibile procedere come esemplificato, generando prima l'albero sintattico e applicando poi, con modalità bottom-up, le regole di traduzione, è certamente preferibile, per motivi di efficienza, fondere il parsing e la traduzione, ossia procedere all'applicazione delle regole sintattiche man mano che il riconoscimento sintattico procede. Con il parsing discendente, ciò non è semplice, in quanto l'albero sintattico è costruito top-down, mentre le regole sintattiche vanno applicate bottom-up.

Il problema può essere risolto modificando il parser predittivo non ricorsivo con l'aggiunta di un secondo stack, detto **stack semantico**, destinato a contenere le traduzioni delle produzioni. Alla fine del processo lo stack semantico conterrà, come unico elemento, la traduzione dell'intera sentenza.

Per illustrare il procedimento, applichiamolo passo-passo al linguaggio delle parentesi tonde e quadre bilanciate, per il quale la traduzione prevista è il numero di coppie di parentesi tonde nella sentenza.

Grammatica	Regole di traduzione
E = ε	E.trans = 0
E = (E)	E_1.trans = E_2.trans + 1
E = [E]	E_1.trans = E_2.trans

Il primo passo è la sostituzione delle regole di traduzione con delle **azioni di traduzione** che si fanno carico di prelevare dallo stack semantico le traduzioni di tutti i non terminali destri e di calcolare e

immettere nello stack semantico la traduzione del non terminale sinistro.

Grammatica	Azioni di traduzione
E = ε	push(0)
E = (E)	push(pop()+1)
E = [E]	~~push(pop())~~

La terza azione è eliminata perché il suo effetto netto è nullo.

Il passo successivo consiste nell'associare a ciascuna azione un numero unico e nell'incorporare questi numeri nella grammatica:

```
Grammatica con azioni
=====================

E = ε #1
E = ( E ) #2
E = [ E ]

#1: push(0)
#2: push(pop()+1)
```

L'algoritmo del parser predittivo è modificato in modo tale che:

1. i numeri di azione sono inseriti nello stack del parser predittivo;
2. quando un numero di azione è estratto dallo stack, si determina l'esecuzione sullo stack semantico delle relative azioni.

Un esempio di esecuzione sulla sentenza **([ ]) $** dell'algoritmo modificato è:

input scandito	stack	stack semantico	azione
(	E$		pop, push((E)#2)
(	(E)#2$		pop, (match), scan
([	E)#2$		pop, push([E])
([	[E])#2$		pop, (match), scan
([]	E])#2$		pop, push(#1)
([]	#1])#2$		pop, do action #1
([]	])#2$	0	pop, (match), scan
([])	)#2$	0	pop, (match), scan
([])$	#2$	0	pop, do action #2
([])$	$	1	pop, (match), scan
([])$			stack vuoto: input accepted!
			traduzione dell'input = 1

Nel caso di produzioni con più di un non terminale a destra, l'azione di traduzione deve effettuare un pop per ogni non terminale. Ad esempio, se la produzione è **`Body = Decls Stmts`** e la traduzione deve dare come risultato il numero di dichiarazioni e statement in ogni body, si ha:

```
Produzione:             Body = Decls Stmts
Regola di traduzione:   Body.trans = Decls.trans + Stmts.trans
Azione di traduzione:   StmtsTrans = pop(); declsTrans = pop();
                        push( stmtsTrans + declsTrans );
Produzione con azione:  Body = Decls Stmts #1
                        #1: StmtsTrans = pop(); DeclsTrans = pop();
                            push( stmtsTrans + declsTrans )
```

Le traduzioni dei non terminali sono estratte dallo stack semantico nell'ordine da destra a sinistra.

In presenza di terminali a destra della produzione, la produzione con azione deve includere, prima del terminale il numero di azione che inserisce al top dello stack semantico il valore del simbolo d'ingresso corrente:

```
Produzione:             F = intero
Regola di traduzione:   F.trans = intero.value
Azione di traduzione:   push( intero.value )
Produzione con azione:  F = #1 intero
                        #1: push( (*ip).value )
```

Quando la grammatica deve essere assoggettata alle trasformazioni necessarie a renderla LL(1), è conveniente partire dalla grammatica originale, incorporare in essa le azioni di traduzione e, solo dopo, procedere ad effettuare le trasformazioni, trattando i numeri di azione come simboli della grammatica. Ad esempio, formata la seguente grammatica non LL(1) con azioni:

```
Grammatica non-LL(1) con azioni
===============================
E = E + T #1
E = T
T = T * F #2
T = F

#1: T_Trans = pop(); E_Trans = pop(); push(E_trans + T_Trans)
#2: F_Trans = pop(); T_Trans = pop(); push(T_trans * F_Trans)
```

si procede ad eliminare la ricorsività a sinistra, ottenendo:

```
Grammatica LL(1) con azioni
===========================
E  = T E'
E' = + T #1 E'
E' = ε
T  = F T'
T' = * F #2 T'
T' = ε
```