Elementi di Sintassi dei Linguaggi di Programmazione

Appunti per gli studenti di Fondamenti di Programmazione
(corsi A-B-C)

Corso di Laurea in Informatica
Università di Pisa
a.a. 2004/05

R. Barbuti, P. Mancarella, D. Pedreschi, F. Turini

Indice

Premessa

Gli studenti sono invitati a leggere queste note con criticità ed attenzione, per la potenziale presenza di refusi ed imprecisioni che possono essere rimasti nel testo. Si ringraziano tutti coloro che vorranno segnalarli; commenti, suggerimenti e critiche riguardanti il materiale o la sua presentazione sono particolarmente benvenuti.

Il materiale di queste note nasce dallo sforzo congiunto di docenti, assistenti e studenti dei corsi di Teoria ed Applicazione delle Macchine Calcolatrici (TAMC) e di Programmazione I dei Corsi di Laurea in Scienze dell'Informazione e in Informatica, per aggiornare e migliorare il corso. Per il loro contributo in tal senso, nonché per le molte osservazioni sul testo, ringraziamo Paola Cappanera, Stefano Cataudella, Alessandra Raffaetà e gli studenti dei corsi di TAMC, Programmazione I e Fondamenti di Programmazione degli Anni Accademici precedenti.

1 La Descrizione della Sintassi

I linguaggi di programmazione, così come tutti i linguaggi usati per la descrizione formale di fenomeni di qualunque tipo, coinvolgono due aspetti distinti, che è importante distinguere: la *sintassi* e la *semantica*. La sintassi ha a che fare con la *struttura* (o la forma) dei programmi esprimibili in un dato linguaggio di programmazione. La semantica, invece, ha a che fare con il *significato* dei programmi esprimibili in un dato linguaggio. Se consideriamo per un momento il linguaggio naturale (italiano) con cui comunichiamo con i nostri simili, due frasi come "*la mela mangia il bambino*" e "*il bambino mangia la mela*" obbediscono entrambe alla sintassi dell'italiano, in quanto sono costruite secondo lo schema <soggetto> <verbo> <complemento-oggetto> a partire da parole costruite correttamente. D'altra parte, solo la seconda ha un significato, o semantica, ragionevole.

In queste note ci concentreremo sul primo aspetto, la descrizione della sintassi dei linguaggi. Il punto di partenza del nostro studio è la concezione del termine *linguaggio* come sinonimo di *insieme di frasi (sintatticamente) ammissibili*. In altre parole, una volta fissato un vocabolario, o *alfabeto*, di elementi di base, detti anche elementi *terminali*, un linguaggio non sarà altro che un sottoinsieme di tutte le frasi ottenibili come sequenze di elementi terminali. Tali sequenze saranno anche riferite con il termine *stringhe*.

Si pensi ad esempio al linguaggio delle espressioni dell'aritmetica, ottenibili a partire da numeri interi e dalle quattro operazioni $+$, $-$, $\times$ e $/$. Fra tutte le possibili sequenze (o stringhe) di numeri ed operazioni aritmetiche, ve ne sono di ammissibili, come $3 \times 4 + 2$, e di non ammissibili, come $3 + \times + 4$. Quindi, il linguaggio delle espressioni aritmetiche può essere identificato con il sottoinsieme delle stringhe ammissibili. In modo del tutto analogo, un linguaggio di programmazione può essere identificato con l'insieme delle proprie stringhe ammissibili, che comunemente chiamiamo *programmi*. Secondo questo punto di vista, descrivere un linguaggio significa sostanzialmente descrivere l'insieme delle stringhe del linguaggio stesso, ovvero avere un metodo per:

- decidere quali stringhe fanno parte di tale insieme, e quali non ne fanno parte, oppure
- costruire tale insieme, enumerando le stringhe che lo compongono.

Il problema insomma è: come identificare l'insieme delle stringhe ammissibili, che caratterizza un linguaggio? Esistono due approcci principali a questo problema. Il primo si basa su uno strumento, detto *automa*, che è in grado di *riconoscere* (o accettare) tutte e sole le stringhe che fanno parte di un linguaggio. Il secondo si basa su uno strumento, detto *grammatica*, che è in grado di *generare* (o costruire) tutte e sole le stringhe che fanno parte di un linguaggio.

Molti studi sono stati dedicati alla definizione e al riconoscimento dei linguaggi. La teoria che è stata sviluppata è conosciuta con il nome di "teoria degli automi" o "teoria dei linguaggi formali" e le sue definizioni di base e tecniche fanno parte del nucleo dell'informatica. Queste note vogliono costituire una introduzione rapida e informale alle idee e ai risultati fondamentali riguardanti gli automi e le grammatiche, al fine di comprendere i metodi con cui si descrive la sintassi dei linguaggi di programmazione studiati nel corso di Programmazione I e, più in generale, nei corsi di studio in Informatica.

1.1 Concetti preliminari

Alla luce delle considerazioni fatte, formalizziamo il concetto di linguaggio.

Abbiamo innanzitutto bisogno di fissare un insieme finito detto *alfabeto*, e denotato con il simbolo Λ (lambda maiuscola). Ciascun elemento dell'alfabeto Λ è detto elemento o simbolo *terminale*.

Abbiamo poi bisogno di definire il concetto di *stringa* sull'alfabeto Λ: una stringa è una sequenza finita di simboli di Λ. Quindi una stringa è una sequenza $a_1 a_2 \cdots a_n$, con $n \geq 0$, dove ciascun a_i è un elemento di Λ: in formule, $\forall i \in [1, n]. a_i \in \Lambda$. Ad esempio, se l'alfabeto Λ è l'insieme $\{0, 1\}$, un esempio di stringa è 00110110001101. Se invece l'alfabeto Λ è l'insieme dei caratteri dell'alfabeto italiano, un esempio di stringa è `ProgrammazioneI`. Consideriano, in generale, una stringa $s = a_1 a_2 \cdots a_n$, con $n \geq 0$. Il numero naturale n è detto *lunghezza* della stringa. Se la lunghezza di una stringa s è 0, allora la stringa s è chiamata *stringa vuota*, ed è rappresentata con il simbolo ϵ. L'insieme di tutte le stringhe su un dato alfabeto Λ è indicato come Λ^*. In generale, il simbolo $*$ è un operatore che, dato un insieme S, costruisce l'insieme di tutte le stringhe sull'insieme S. In formule:

$$S^* = \{s \mid s \text{ è una stringa su } S\}.$$

Si noti che, se Λ, com'è naturale, non è vuoto, allora Λ^* è comunque un insieme infinito, formato cioè da infiniti elementi, anche se ciascuno di questi, ovvero ciascuna stringa, è di lunghezza finita.

L'operazione principale su stringhe è quella di *concatenazione*: date due stringhe s e t, la loro concatenazione è rappresentata come st, ed il risultato dell'operazione è una nuova stringa composta dai simboli di s seguiti da quelli di t. Formalmente, se $s = a_1 a_2 \cdots a_n$ e $t = b_1 b_2 \cdots b_m$, con $n, m \geq 0$, allora:

$$st = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m.$$

Ad esempio, se $s = 0011$ e $t = 1100$, allora $st = 00111100$.

Infine, non ci resta che da precisare che cos'è un *linguaggio* (su un alfabeto Λ): questo, per i nostri scopi, non sarà altro che un sottoinsieme di Λ^*, ovvero un insieme di stringhe sull'alfabeto. In formule, L è un linguaggio su Λ se

$$L \subseteq \Lambda^*.$$

Secondo questa definizione, Λ^* stesso e l'insieme vuoto $\emptyset$ sono due esempi di linguaggi, per la verità entrambi poco interessanti. I linguaggi "interessanti", su un dato alfabeto Λ, saranno infatti insiemi non vuoti e non banali di stringhe che hanno in comune una qualche proprietà rilevante. Sui linguaggi, visti come insiemi di stringhe, sono definite tutte le operazioni insiemistiche: unione ($\cup$), intersezione ($\cap$), complemento e quelle da esse derivate. E' definita inoltre l'operazione di concatenazione, basata naturalmente sulla concatenzione di stringhe, nel seguente modo: siano $L_1, L_2 \subseteq \Lambda^*$

$$L_1 L_2 = \{\alpha\beta \mid \alpha \in L_1 \wedge \beta \in L_2\}$$

Secondo questa visione, i linguaggi di programmazione sono essenzialmente sottoinsiemi di $\mathtt{ASCII}^*$, dove `ASCII` è l'alfabeto dei caratteri alfabetici, numeri e di interpunzione presenti, secondo uno standard internazionale, sulla tastiera alfanumerica di ogni computer. In altre parole, un programma non è altro che una stringa di caratteri alfanumerici `ASCII`. Ovviamente, non tutte le stringhe siffatte sono programmi accettabili, così come i diversi linguaggi di programmazione corrispondono a diversi insiemi di tali stringhe. Ogni linguaggio di programmazione ha la propria sintassi, ovvero le proprie regole che descrivono la struttura delle stringhe `ASCII` che corrispondono a programmi ammissibili, o sintatticamente ben formati.

Il resto di queste note è dedicato alla presentazione di due metodi, gli automi e le grammatiche, rivolti proprio a descrivere la struttura sintattica dei linguaggi.

2 Gli Automi

In questa sezione saranno affrontati gli aspetti seguenti.

- Un *automa finito* è un metodo, basato su diagrammi detti *grafi*, per specificare linguaggi; ci sono due tipi di automi: deterministici (Paragrafo 2.1) e non deterministici (Paragrafo 2.2).
- Un automa non deterministico può essere trasformato in un automa deterministico che riconosce lo stesso linguaggio, usando la tecnica di "costruzione dei sottoinsiemi" (Paragrafo 2.3).

2.1 Macchine a stati e automi

I riconoscitori che cercano di riconoscere le stringhe di un linguaggio hanno spesso una struttura particolare, quella di una *macchina a stati*: è possibile in genere individuare certi punti in cui possiamo affermare di conoscere lo stato di avanzamento della macchina nel processo di riconoscimento di una stringa. Chiamiamo questi punti *stati*: possiamo pensare che la macchina funzioni passando da uno stato ad un altro, in conseguenza della lettura dei dati in ingresso, ovvero degli elementi della stringa, uno dopo l'altro.

Per concretizzare questa idea, consideriamo un problema specifico di riconoscimento di stringhe: "quali parole inglesi contengono le cinque vocali in ordine alfabetico"? Per rispondere a questa domanda, possiamo utilizzare la lista di parole fornita da molti sistemi operativi; per esempio, nel sistema UNIX è possibile trovarla nel file `/usr/dict/words`, che contiene le parole inglesi più comuni, una per linea. Alcune delle parole di questo file che contengono le cinque vocali in ordine, sono: `facetious, sacrilegious.` Si noti come anche la terza parola va bene secondo la formulazione del problema: basta ignorare la prima occorrenza da sinistra della vocale `i` in `sacrilegious`. Esaminiamo come una macchina molto semplice può trovare tutte le parole che contengono le cinque vocali in ordine. La macchina può leggere ciascuna parola ed esaminarne i caratteri. Iniziando l'esame della parola da sinistra verso destra la macchina cerca una `a`. Diciamo che la macchina si trova nello "stato 0" fintanto che non trova una `a`, mentre quando la trova passa nello "stato 1". Nello stato 1, la macchina cerca una `e` e, quando ne trova una, passa nello "stato 2". Si procede in questo modo fino a raggiungere lo "stato 4" in cui si cerca una `u`. Se viene trovata una `u`, allora la parola contiene le vocali, ordinate alfabeticamente, e la macchina può terminare nello stato di riconoscimento, "stato 5", in cui riconosce la parola. Esaminando il resto della parola, la macchina continua a rimanere nello stato 5 in presenza di ogni nuovo carattere letto, dato che sappiamo che la stringa in questione è riconosciuta indipendentemente da ciò che segue `u`.

Possiamo interpretare lo stato i come l'indicazione che la macchina ha già trovato le prime i vocali, in ordine alfabetico, per $i = 0, 1, \ldots, 5$. I sei stati riassumono tutto quello che il programma deve ricordare durante l'esame della parola, da sinistra a destra. Per esempio, nello stato 0, quando la macchina cerca una `a` non ha bisogno di ricordare se ha già incontrato una `e`. La ragione è che questa `e` non è preceduta da nessuna `a` e quindi non può servire come `e` nella sottosequenza `aeiou`.

Formalmente un automa è definito come una quintupla

$$\langle \Lambda, \Sigma, S, F, \delta \rangle$$

dove

- Λ è l'alfabeto su cui è definito il linguaggio riconosciuto dall'automa;
- Σ è l'insieme **finito** degli stati dell'automa;
- $S \in \Sigma$ è lo stato iniziale, ovvero lo stato in cui si trova l'automa quando inizia il tentativo di riconoscimento;
- $F \subseteq \Sigma$ è il sottoinsieme degli stati finali, ovvero quegli stati in cui l'automa, dopo aver analizzato l'intera stringa, si arresta con riconoscimento della stessa;
- $\delta \subseteq (\Sigma \times \Lambda) \times \Sigma$ è la relazione di transizione che associa una coppia

$$(s, a) \in \Sigma \times \Lambda$$

ad uno stato $s' \in \Sigma$. L'appartenenza della coppia $\langle\langle s_i, a\rangle, s_j\rangle$ a δ significa che se l'automa si trova nello stato s_i e il simbolo da analizzare è a, allora si sposta sul simbolo successivo della stringa e nello stato s_j. E' importante osservare che δ è una relazione di transizione, ovvero è possibile che, dato uno stato di partenza e un simbolo, ci possano essere più stati in cui effettuare la transizione.

Grafi che rappresentano macchine a stati

È possibile rappresentare il funzionamento di una macchina a stati come quella descritta nel paragrafo precedente mediante un diagramma costituito da *nodi* e *archi*. Graficamente, i nodi sono rappresentati mediante cerchi, e gli archi da frecce che congiungono due nodi: un nodo di partenza ed un nodo di arrivo dell'arco. Un diagramma di questo tipo è chiamato *grafo diretto*, ed è una struttura che incontreremo spesso durante il corso di studi in informatica. Spesso, i nodi e gli archi sono *etichettati*, ovvero è loro associata una informazione, o etichetta.

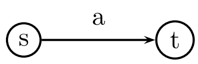
Nel diagramma, o grafo, che rappresenta una macchina a stati, i nodi rappresentano gli stati, e gli archi le transizioni fra stato e stato.

In tale diagramma, gli archi sono etichettati da simboli dell'alfabeto e i nodi da stati dell'automa.

Se alla relazione di transizione δ appartiene la coppia

$$\langle\langle s, a\rangle, t\rangle$$

nel diagramma compare



Per comodità rappresenteremo più transizioni con lo stesso stato di partenza e di arrivo, ma con simboli diversi con un unico arco etichettato da un insieme di simboli.

Alcuni nodi sono distinti come *stati di riconoscimento*, o *stati di accettazione*, o anche *stati finali*: quando l'esame di tutti i caratteri di una stringa termina in uno di questi stati, abbiamo riconosciuto la stringa come appartenente al linguaggio, e la "accettiamo". Per convenzione, gli stati di riconoscimento sono rappresentati nel diagramma da doppi cerchi.

Infine uno dei nodi è indicato come *stato iniziale*, da cui ha inizio la procedura di riconoscimento di una stringa. Lo stato iniziale è individuato da una freccia che arriva a quel nodo ma che non proviene da nessun altro nodo. Un grafo di questo tipo è chiamato *automa a stati finiti* (o anche, più correttamente, automa con un numero finito di stati), o semplicemente *automa*; un esempio è dato in Figura 1.

Il funzionamento di un automa è concettualmente semplice: dopo aver ricevuto una successione di caratteri, detta *stringa (o sequenza) di ingresso*, l'automa inizia dallo stato iniziale leggendo il primo carattere di tale sequenza. Sulla base del carattere esaminato viene eseguita una transizione ad un nuovo stato, che può anche coincidere con quello in cui l'automa già si trova. La transizione è definita dal grafo dell'automa. A questo punto l'automa legge il secondo carattere, esegue la transizione opportuna e così via.

Esempio 1 L'automa corrispondente alla macchina che riconosce stringhe che contengono `aeiou` come sottosequenza è mostrato in Figura 1. Al solito, con la lettera greca Λ (lambda maiuscola) indichiamo l'insieme di tutte le lettere, maiuscole e minuscole, dell'alfabeto inglese. Usiamo anche abbreviazioni come $\Lambda - \mathtt{a}$ (al posto di $\Lambda \setminus \{\mathtt{a}\}$) per rappresentare l'insieme di tutte le lettere escluso `a`.

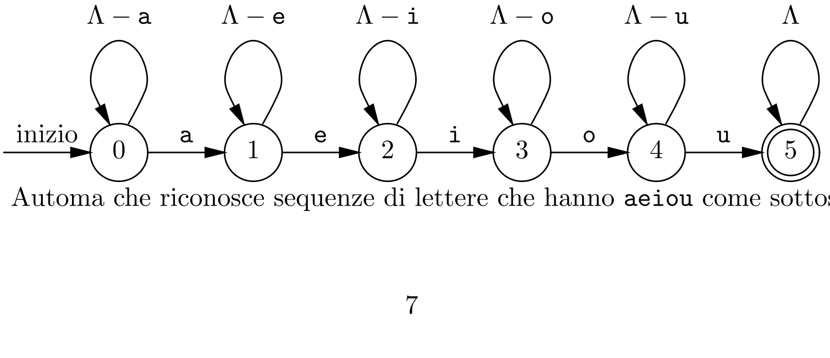


Figura 1: Automa che riconosce sequenze di lettere che hanno `aeiou` come sottosequenza

Il nodo 0 è lo stato iniziale: in presenza di una qualsiasi lettera diversa da `a` rimaniamo nello stato 0, ma non appena incontriamo una `a` andiamo nello stato 1. In modo simile, una volta raggiunto lo stato 1, rimaniamo in attesa di una `e` e, quando questa si presenta, andiamo nello stato 2. I due stati successivi, 3 e 4, vengono raggiunti quando si incontrano una `i` e quindi una `o`. Restiamo nello stato 4 fino a che non troviamo una `u`; quando la troviamo passiamo nello stato 5, che è l'unico stato di riconoscimento. Nello stato 5 continuiamo a restare in presenza di qualunque ulteriore lettera, perché la stringa sarà a questo punto riconisciuta indipendentemente da ciò che rimane da esaminare della stringa in ingresso.

Val la pena di notare che, se negli stati da 0 a 4 incontriamo un carattere spazio (o un qualsiasi carattere diverso da una lettera), non abbiamo nessuna transizione: in questo caso il procedimento si arresta e, poiché non abbiamo esaurito la stringa in ingresso, abbiamo "rifiutato" l'ingresso. Analogamente, se l'esame della stringa in ingresso termina in uno stato diverso dal 5, il riconoscimento della stringa fallisce.

Esercizi

2.1.1

Progettare degli automi, con stringhe di 0 e 1 come ingresso, che eseguano le funzioni seguenti:

a) determinare se la sequenza letta ha parità pari (cioè contiene un numero pari di 1): l'automa accetta una stringa se questa ha parità pari e la rifiuta se ha parità dispari;

b) controllare che non ci siano più di due 1 consecutivi: l'automa accetta una stringa solo se 111 non è una sua sottostringa;

Qual è il significato intuitivo di ciascuno stato degli automi così costruiti?

2.1.2

Indicare la sequenza di stati e le uscite quando viene dato l'ingresso 101001101110 ai due automi dell'Esercizio 2.1.1.

2.1.3

Progettare un automa che legge una parola (stringa di caratteri) e determina se le lettere della parola sono ordinate. Per esempio, `acino` e `bello` hanno le lettere in ordine alfabetico, mentre `ballo` non le ha, dato che la lettera `a` segue la `b`. La parola deve terminare con un carattere spazio, in modo che l'automa sappia quando ha letto tutti i caratteri. Quanti stati sono necessari? Qual è il loro significato intuitivo? Quante transizioni escono da ogni stato? Quanti sono gli stati di riconoscimento?

2.1.4

Progettare un automa che determina se una stringa di caratteri è un pronome di terza persona singolare della lingua inglese: `he`, `his`, `him`, `she`, `her` o `hers`, seguito da uno spazio.

2.2 Automi deterministici e non deterministici

Dato un grafo che rappresenta un automa, è importante chiarire il concetto di *cammino* sul grafo, che permette di formalizzare il comportamento di un automa mentre esamina una stringa in ingresso. Intuitivamente, un cammino è una sequenza di nodi del grafo, percorribile usando gli archi del grafo stesso. Ad esempio, in riferimento al grafo di Figura 1, la sequenza di stati 0112334 è un cammino, mentre 432 non lo è. Più formalmente, un *cammino* su un grafo G è una sequenza $n_1 n_2 \cdots n_k$ di nodi di G tale che, per ogni $i \in [1, k-1]$, esiste un arco in G dal nodo n_i al nodo n_{i+1}. Se, come nel nostro caso, il grafo è etichettato, ad ogni cammino corrispondono una o più stringhe, dette *etichette* del cammino, ciascuna delle quali si ottiene concatenando, nell'ordine, una fra le etichette di ciascun arco attraversato dal cammino.

L'operazione di base nell'uso degli automi è quella di prendere una stringa in ingresso $a_1 a_2 \cdots a_k$ e seguire, a partire dallo stato iniziale, un cammino i cui archi sono etichettati con i simboli della sequenza, nell'ordine; cioè, per $i = 1, 2, \ldots, k$, il simbolo a_i appartiene all'insieme S_i che etichetta l'i-esimo arco del cammino. La costruzione di questo cammino insieme alla sua sequenza di stati costituisce la *simulazione* dell'automa sulla sequenza di ingresso $a_1 a_2 \cdots a_k$. Questo cammino ha $a_1 a_2 \cdots a_k$ come *etichetta*, ma, ovviamente, può avere anche altre etichette, dato che gli insiemi S_i che etichettano gli archi lungo il cammino possono contenere altri caratteri.

Esempio 2 Consideriamo l'automa di Figura 1, che abbiamo usato per riconoscere le parole con sottosequenza `aeiou`. Consideriamo la stringa di caratteri `adept`.

Partiamo dallo stato 0: ci sono due transizioni che partono dallo stato 0, una per l'insieme di caratteri $\Lambda - \texttt{a}$ e un'altra per `a` da solo. Poiché il primo carattere in `adept` è `a`, seguiamo la seconda transizione che porta nello stato 1. Dallo stato 1 partono due transizioni etichettate, rispettivamente, da $\Lambda - \texttt{e}$ e `e`; poiché il secondo carattere della stringa è `d`, dobbiamo eseguire la prima dato che tutte le lettere, esclusa la `e`, appartengono all'insieme $\Lambda - \texttt{e}$. Questa transizione ci lascia nello stato 1. Visto che il terzo carattere è `e`, eseguiamo la seconda transizione dallo stato 1, che ci porta nello stato 2. Le due lettere finali di `adept` sono entrambe incluse nell'insieme $\Lambda - \texttt{i}$ e quindi le ultime due transizioni vanno dallo stato 2 allo stato 2. Terminiamo dunque l'esame di `adept` nello stato 2. Dal momento che lo stato 2 non è uno stato di riconoscimento, non accettiamo l'ingresso `adept`. Si noti che il cammino corrispondente a questa stringa è 011222.

Automi deterministici

Gli automi discussi nel paragrafo precedente hanno una proprietà importante: per ogni stato s e ogni carattere di ingresso x, c'è al più una transizione dallo stato s la cui etichetta contiene x. Un automa di questo tipo è detto *deterministico*.

Simulare un automa deterministico su una sequenza di ingresso data è molto semplice. In ogni stato s, dato il prossimo carattere di ingresso x, consideriamo le etichette delle transizioni che partono da s. Se troviamo una transizione la cui etichetta include x, allora quella transizione porta al prossimo stato. Se nessuna etichetta include x, allora l'automa "muore" o "fallisce" e non può esaminare alcun altro ingresso, come nel caso dell'Esempio 2.

Formalizziamo la nozione di accettazione, o riconoscimento, di una stringa (su un dato alfabeto Λ) da parte di un automa. Diremo che un automa *accetta* (o *riconosce*) una stringa s se:

- esiste un cammino sul grafo dell'automa etichettato con s, e
- l'ultimo stato di tale cammino è uno stato finale.

Secondo questa definizione, un automa può fallire nel riconoscimento di una stringa per due motivi distinti: (a) perché non riesce a completare l'esame della stringa per mancanza di opportune transizioni (archi) sul grafo, oppure (b) perché l'esame della stringa termina in uno stato non finale. È quindi semplice definire il linguaggio accettato o riconosciuto da un automa come quel particolare sottoinsieme di Λ^* costituito dalle stringhe accettate dall'automa stesso. Dato un automa G, definiamo infatti il linguaggio riconosciuto da G, in simboli $L(G)$, nel modo seguente:

$$L(G) = \{s \in \Lambda^* \mid s \text{ è accettata da } G\}.$$

In riferimento all'esempio di Figura 1, è possibile dimostrare facilmente che il linguaggio riconosciuto dall'automa è:

$$\{s_1\, \texttt{a}\, s_2\, \texttt{e}\, s_3\, \texttt{i}\, s_4\, \texttt{o}\, s_5\, \texttt{u}\, s_6 \mid s_1, s_2, s_3, s_4, s_5, s_6 \in \Lambda^* \}.$$

Nella definizione di "automa" non è però richiesto che le etichette sulle transizioni che escono da uno stato siano disgiunte (due insiemi sono *disgiunti* se non hanno elementi comuni, cioè se la loro intersezione è vuota). Prendiamo ad esempio il grafo mostrato in Figura 2, dove per l'ingresso x ci sono transizioni dallo stato s agli stati t e u: non è chiaro come questo automa possa procedere al riconoscimento di una stringa, potendo in generale seguire strade diverse.

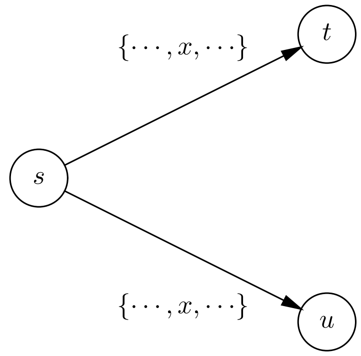


Figura 2: Transizione non deterministica dallo stato s in presenza dell'ingresso x

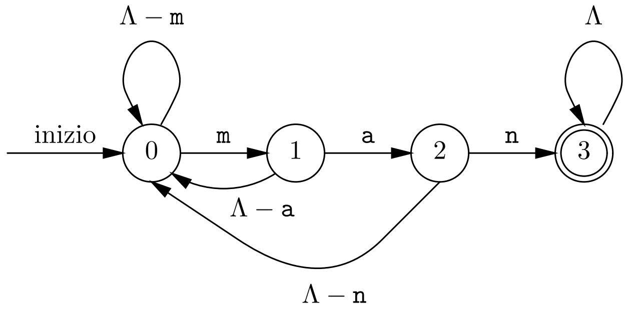


Figura 3: Automa deterministico che riconosce molte stringhe che contengono `man`, ma non tutte.

Automi non deterministici

Gli automi *non deterministici* possono avere (ma non necessariamente hanno) due o più transizioni, a partire dallo stesso stato, che contengono lo stesso simbolo. Si noti che un automa deterministico è anche un particolare automa non deterministico, uno in cui c'è sempre una sola transizione possibile per un simbolo di ingresso. Un "automa" è in generale non deterministico, tuttavia useremo il termine "automa non deterministico" quando vorremo sottolineare il fatto che l'automa non deve essere necessariamente deterministico. Usando la "costruzione dei sottoinsiemi", che tratteremo nel prossimo paragrafo, è possibile trasformare ogni automa non deterministico in uno deterministico che riconosce le stesse sequenze di caratteri.

Quando proviamo a simulare un automa non deterministico su una stringa di caratteri in ingresso, $a_1 a_2 \cdots a_k$, possiamo scoprire che questa stringa etichetta molti cammini. La nostra definizione di accettazione di una stringa da parte di un automa si applica anche nel caso non deterministico, ed implica che l'automa *accetta* la stringa di ingresso se *almeno uno* dei cammini etichettati dalla stringa porta all'accettazione: anche un solo cammino che finisca in uno stato di riconoscimento è sufficiente, indipendentemente da quanti siano i cammini che terminino in uno stato di non riconoscimento.

Esempio 3 La Lega Inglese Contro i Discorsi Sessisti (*League Against Sexist Speech*, LASS) vuole scoprire tutti gli scritti sessisti che contengono la parola "man" (uomo). I componenti della Lega, non solo vogliono scoprire costruzioni come "chairman" (presidente), ma anche forme più sottili di discriminazione come nelle parole "maniac" (maniaco) o "emancipate" (emancipato). La LASS vuole progettare un programma usando un automa; il programma deve esaminare stringhe di caratteri e "accettare" quelle che contengono la stringa di caratteri `man`, ovunque essa compaia.

Si può pensare, per prima cosa, ad un automa come quello di Figura 3. Lo stato 0 rappresenta il caso in cui non abbiamo ancora cominciato a vedere le lettere di "man". Lo stato 1 vuol rappresentare la situazione in cui abbiamo visto una `m`; nello stato 2 abbiamo già visto `ma` e, nello stato 3, abbiamo

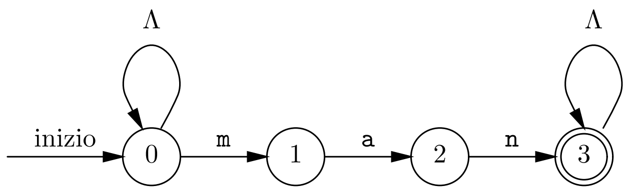


Figura 4: Automa non deterministico che riconosce le stringhe che contengono `man`

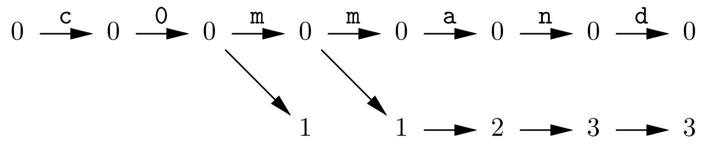


Figura 5: Simulazione dell'automa non deterministico di Figura 4 sulla stringa di ingresso `command`.

visto `man`. Negli stati 0, 1 e 2, se non vediamo in ingresso la lettera sperata (`m`, `a` e `n` rispettivamente), torniamo allo stato 0 e proviamo ancora.

Tuttavia, l'automa di Figura 3 non funziona correttamente. Sulla stringa di ingresso `command`, rimane nello stato 0 leggendo `c` e `o`. Passa nello stato 1 leggendo la prima `m`, ma la seconda `m` lo riporterà allo stato 0, che non verrà più abbandonato.

La Figura 4 mostra un automa non deterministico che riconosce tutte le stringhe di caratteri che contengono la sottostringa `man`. La chiave del successo è che, nello stato 0, supponiamo contemporaneamente che una `m` sia l'inizio di `man` e che non lo sia. Dato che l'automa è non deterministico può supporre allo stesso tempo sia di "sì" (tentativo rappresentato dalla transizione dallo stato 0 allo stato 1) che di "no" (tentativo rappresentato dal fatto che la transizione dallo stato 0 allo stato 0 può essere effettuata con tutte le lettere, `m` inclusa). Poiché il riconoscimento mediante un automa non deterministico richiede un solo cammino che raggiunge uno stato di accettazione, possiamo avere i vantaggi di entrambi i tentativi.

La Figura 5 mostra le azioni dell'automa non deterministico di Figura 4 sulla stringa di ingresso `command`. In corrispondenza dei caratteri `c` e `o`, l'automa può solamente rimanere nello stato 0. Quando si presenta il primo `m` l'automa può scegliere di andare allo stato 0 o allo stato 1; vengono quindi esplorate entrambe le scelte. Con il secondo `m` non c'è nessuno stato da raggiungere dallo stato 1 e quindi questo cammino "muore". Tuttavia, dallo stato 0 possiamo ancora andare nello stato 0 e nello stato 1. Quando si presenta l'ingresso `a`, possiamo passare dallo stato 0 allo stato 0 e dallo stato 1 allo stato 2. Analogamente, con l'ingresso `n` possiamo passare dallo stato 0 allo stato 0 e dallo stato 2 allo stato 3.

Poiché lo stato 3 è uno stato di accettazione e tutte le possibili transizioni sono ammesse in 3, la stringa è riconosciuta. Il fatto che l'automa si trova anche nello stato 0 dopo aver esaminato `comman` è irrilevante ai fini del riconoscimento. La transizione finale, con ingresso `d`, va dallo stato 0 allo stato 0. Si noti che la transizione che riporta allo stato 0, presente in Figura 3 per gestire il caso in cui non si presenta la prossima lettera della parola `man`, non è necessaria in Figura 4, dato che, in questo caso, non siamo forzati a percorrere la sequenza da 0 a 1 e da 2 a 3. Si suppone contemporaneamente, invece, sia che non stia per iniziare la sequenza `man`, rimanendo nello stato 0, sia che ogni `m` sia l'inizio di `man`.

Vedremo, nel prossimo paragrafo, come sia possibile trasformare l'automa di Figura 4 in un automa deterministico. Questo automa deterministico, diversamente da quello di Figura 3, riconosce correttamente tutte le occorrenze di `man`.

Vedremo come sia possibile convertire sempre ogni automa non deterministico in uno deterministico. Esistono però automi non deterministici per i quali il corrispondente automa deterministico ha molti più stati: in generale, un automa deterministico con n stati può essere trasformato in un automa deterministico con 2^n stati. Da questo fatto possiamo trarre la conclusione che, per risolvere un dato problema, un automa non deterministico può essere notevolmente più semplice da progettare di uno deterministico.

Esercizi

2.2.1

La LASS vuole trovare tutte le occorrenze delle stringhe `man`, `son` (figlio) e `father` (padre). Progettare un automa non deterministico che riconosce le parole che contengono almeno una di queste tre stringhe di caratteri.

2.2.2

Progettare un automa deterministico che risolve il problema dell'esercizio precedente.

2.2.3

Simulare gli automi delle Figure 3 e 4 sulla stringa di ingresso `summand`.

2.3 Dal non determinismo al determinismo

In questo paragrafo vedremo come ogni automa non deterministico può essere rimpiazzato da uno deterministico equivalente. Come abbiamo visto, per risolvere un problema è spesso più semplice pensare ad un automa non deterministico che a uno deterministico; d'altra parte, il procedimento di riconoscimento di un automa deterministico è più semplice di quello non deterministico, in quanto per riconoscere una stringa non è necessario esplorare una moltitudine di possibili cammini. Quindi, un automa deterministico può essere più facilmente realizzato in pratica, ed è quindi importante che ci sia un algoritmo per trasformare automi non deterministici in automi deterministici equivalenti.

Equivalenza di automi

Formalmente, supposto che A e B siano due automi (deterministici o meno), diciamo che A e B sono *equivalenti* se accettano lo stesso insieme di stringhe di ingresso. Ovvero, se $a_1a_2\cdots a_k$ è una qualsiasi sequenza di simboli, allora entrambe le condizioni seguenti sono verificate.

1. Se c'è un cammino etichettato da $a_1a_2\cdots a_k$ dallo stato iniziale di A a uno stato di accettazione di A, allora c'è anche un cammino etichettato da $a_1a_2\cdots a_k$ dallo stato iniziale di B a uno stato di accettazione di B.

2. Se c'è un cammino etichettato da $a_1a_2\cdots a_k$ dallo stato iniziale di B a uno stato di accettazione di B, allora c'è anche un cammino etichettato da $a_1a_2\cdots a_k$ dallo stato iniziale di A a uno stato di accettazione di A.

Esempio 4 Consideriamo gli automi delle Figure 3 e 4. Come abbiamo notato nella Figura 5, l'automa di Figura 4 accetta la stringa di ingresso `comman` poiché questa sequenza di caratteri etichetta il cammino $0 \to 0 \to 0 \to 0 \to 1 \to 2 \to 3$ in Figura 4, e questo cammino va dallo stato iniziale ad uno stato di riconoscimento. Tuttavia, nell'automa di Figura 3, che è deterministico, possiamo vedere che il solo cammino etichettato da `comman` è $0 \to 0 \to 0 \to 1 \to 0 \to 0 \to 0$. Quindi, se chiamiamo A l'automa di Figura 3 e B quello di Figura 4, il punto (2) non è verificato, con la conseguenza che i due automi non sono equivalenti.

Costruzione dei sottoinsiemi

Vedremo adesso come "eliminare il non determinismo da un automa", costruendo un automa deterministico equivalente. Questa tecnica è chiamata *costruzione dei sottoinsiemi* e il suo spirito è suggerito dalla Figura 5, in cui simuliamo un automa non deterministico su un particolare ingresso. Notiamo, da queste figure, che, ad ogni istante, l'automa si trova in un insieme di stati e che questi stati compaiono in una colonna del diagramma di simulazione. Cioè, dopo aver letto l'ingresso $a_1a_2\cdots a_k$, l'automa non deterministico è "in" quegli stati che vengono raggiunti dallo stato iniziale attraverso i cammini etichettati da $a_1a_2\cdots a_k$.

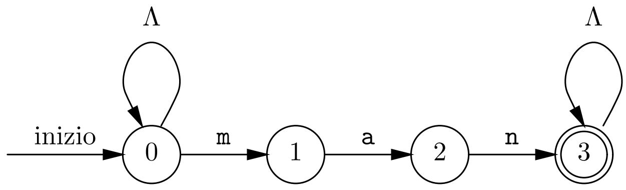


Figura 6: Automa non deterministico che riconosce le parole che contengono `man`

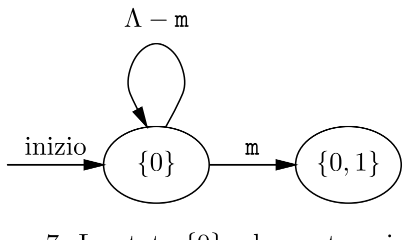


Figura 7: Lo stato $\{0\}$ e le sue transizioni

Adesso abbiamo un indizio su come trasformare un automa non deterministico N in uno deterministico D. Ogni stato di D sarà un insieme di stati di N, mentre le transizioni tra gli stati di D saranno determinate dalle transizioni di N. Per vedere come sono costruite le transizioni di D, consideriamo uno stato S di D e un simbolo di ingresso x. Siccome S è uno stato di D, è composto da stati di N; definiamo l'insieme T come l'insieme degli stati t dell'automa N tali che esiste uno stato s appartenente a S e una transizione in N da s a t, etichettata da un insieme che contiene il simbolo di ingresso x. Costruiamo, nell'automa D, una transizione da S a T etichettata dal simbolo x.

Sappiamo ora come costruire una transizione tra due stati dell'automa deterministico D, ma dobbiamo ancora determinare l'insieme degli stati, lo stato iniziale e gli stati di riconoscimento di D. Costruiamo gli stati di D per induzione.

Base. Se lo stato iniziale di un automa non deterministico N è s_0, allora lo stato iniziale dell'automa deterministico D è $\{s_0\}$, cioè l'insieme contenente solo s_0.

Induzione. Supponiamo di aver scoperto che S, un insieme di stati di N, è uno stato di D. Consideriamo, in sequenza, tutti i possibili caratteri di ingresso x. Per un dato x, chiamiamo T l'insieme degli stati t di N tali che da uno stato s in S esiste una transizione da s a t con ingresso x. Allora l'insieme T è uno stato di D ed esiste una transizione da S a T con ingresso x.

Gli stati di accettazione di D sono quegli insiemi di stati di N che includono almeno uno stato di accettazione di N. Tutto questo è molto intuitivo: se S è uno stato di D, e quindi un insieme di stati di N, allora l'ingresso $a_1a_2\cdots a_k$ che porta D dal suo stato iniziale allo stato S, porterà anche N dal suo stato iniziale a tutti gli stati in S. Se S include uno stato di accettazione, $a_1a_2\cdots a_k$ è accettata da N e anche D deve accettarla. Poiché D, leggendo l'ingresso $a_1a_2\cdots a_k$, giunge soltanto a S, quest'ultimo deve essere uno stato di accettazione.

Esempio 5 Trasformiamo l'automa non deterministico di Figura 4, che riportiamo in Figura 6, in uno deterministico D. Cominciamo con $\{0\}$, che è lo stato iniziale di D.

La parte induttiva della costruzione ci obbliga a guardare ogni stato di D e determinare le sue transizioni. Per $\{0\}$ dobbiamo solo chiederci quali sono le transizioni per 0. La risposta, che otteniamo osservando la Figura 6, è che con tutte le lettere, esclusa la `m`, dallo stato 0 si passa soltanto nello stato 0, mentre con `m` si passa sia in 0 che in 1. L'automa D perciò deve avere lo stato $\{0\}$, che già ha, e lo stato $\{0,1\}$, che deve essere aggiunto. Le transizioni e gli stati di D costruiti fino ad adesso sono mostrati in Figura 7.

Dobbiamo adesso considerare le transizioni che escono da $\{0,1\}$. Esaminando ancora la Figura 6, vediamo che tutte le lettere di ingresso, escluso `m` e `a`, permettono di passare dallo stato 0 allo stato 0, mentre dallo stato 1 non è possibile passare ad alcun altro stato. Quindi costruiamo una transizione

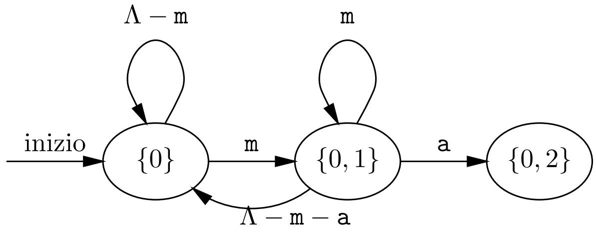


Figura 8: Gli stati $\{0\}$ e $\{0, 1\}$ con le loro transizioni

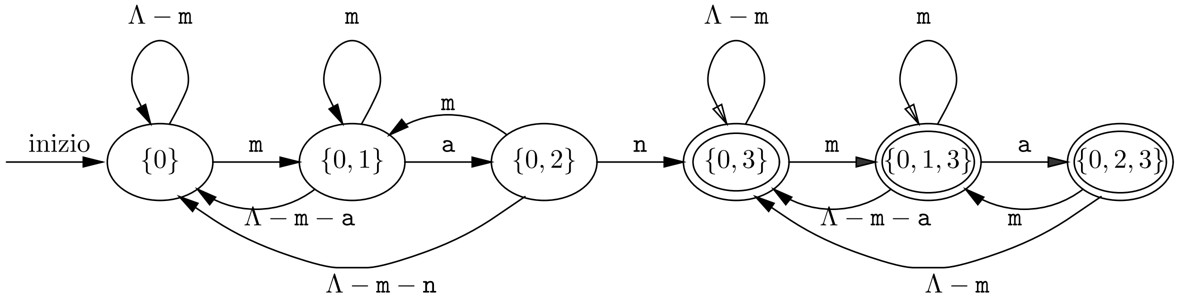


Figura 9: L'automa deterministico D corrispondente a quello non deterministico di Figura 6

dallo stato $\{0, 1\}$ allo stato $\{0\}$, etichettata da tutte le lettere escluso `m` e `a`. Con ingresso `m`, dallo stato 1 non si esegue nessuna transizione, mentre da 0 si può passare sia in 0 che in 1: quindi aggiungiamo una transizione da $\{0, 1\}$ in se stesso etichettata da `m`. Infine, con l'ingresso `a`, dallo stato 0 si può solo rimanere in 0, mentre dallo stato 1 si passa nello stato 2: avremo quindi una transizione etichettata `a` dallo stato $\{0, 1\}$ allo stato $\{0, 2\}$. La porzione di D costruita finora è riportata in Figura 8.

Dobbiamo ora costruire le transizioni per lo stato $\{0, 2\}$. Su tutti gli ingressi, escluso `m` e `n`, dallo stato 0 si passa soltanto in 0 e dallo stato 2 non è possibile andare in alcun altro stato; così c'è una transizione da $\{0, 2\}$ a $\{0\}$ etichettata con tutte le lettere escluso `m` e `n`. Con ingresso `m`, dallo stato 2 non è possibile eseguire alcuna transizione, mentre da 0 si passa sia in 0 che in 1: avremo quindi una transizione da $\{0, 2\}$ a $\{0, 1\}$, etichettata `m`. Con ingresso `n`, dallo stato 0 si rimane in 0 e da 2 si passa in 3. Quindi c'è una transizione etichettata `n` da $\{0, 2\}$ a $\{0, 3\}$. Si tratta di uno stato di accettazione di D, poiché include lo stato di accettazione 3 dell'automa di Figura 6.

Continuando in questo modo, si definiscono i nuovi stati $\{0, 3\}$, $\{0, 1, 3\}$, $\{0, 2, 3\}$, e dato che le transizioni da $\{0, 2, 3\}$ non portano a nessun nuovo stato di D, la costruzione dell'automa è terminata. La definizione completa dell'automa deterministico è mostrata in Figura 9.

Si noti che questo automa deterministico accetta tutte e sole le stringhe di lettere che contengono `man`. Intuitivamente, l'automa si trova nello stato $\{0\}$ quando la stringa di caratteri esaminata non termina con nessun prefisso di `man`, a parte la stringa vuota. Lo stato $\{0, 1\}$ significa che la stringa esaminata finora termina con `m`, $\{0, 2\}$ significa che termina con `ma` e $\{0, 3\}$ significa che termina con `man`.

Formalmente l'automa deterministico equivalente ad un automa non deterministico dato può essere definito come segue:

Sia

$$M = \langle \Lambda, \Sigma, S, F, \delta \rangle$$

un automa a stati finiti non deterministico.

L'automa deterministico equivalente è il seguente:

$$M' = \langle \Lambda, \Sigma', S', F', \delta' \rangle$$

dove:

- Λ è ovviamente lo stesso;
- $\Sigma' = \mathcal{P}_\Sigma$ ovvero è l'insieme dei sottoinsiemi di Σ;
- $S' = \{S\}$ ovvero è l'insieme che contiene solo lo stato iniziale dell'automa a stati finiti non deterministico M;
- $F' = \{A \mid A \in \mathcal{P}_\Sigma \wedge \exists s \in F.\ s \in A\}$ ovvero è l'insieme dei sottoinsiemi di Σ che contiene almeno uno stato finale dell'automa a stati finiti non deterministico M;
- $\delta' : \mathcal{P}_\Sigma \times \Lambda \to \mathcal{P}_\Sigma$

$$\delta'(\{s_1, \ldots, s_n\}, a) = \bigcup_{i=1}^{n} \{t \mid \langle\langle s_i, a\rangle, t\rangle \in \delta\}$$

  ovvero è quella relazione che associa allo stato $\{s_1, \ldots, s_n\}$ di M' e al simbolo a, lo stato di M' costituito da tutti gli stati di M a cui si arriva leggendo a da uno qualsiasi degli stati $s_1, \ldots, s_n$ di M.

Perché la costruzione dei sottoinsiemi funziona

Chiaramente, se D è costruito a partire da un automa non deterministico N usando la costruzione dei sottoinsiemi, allora D è un automa deterministico. La ragione è che, per ciascun simbolo di ingresso x e ciascuno stato S di D, costruiamo un solo stato T di D tale che l'etichetta della transizione da S a T contiene x. Ma come facciamo a sapere che gli automi N e D sono equivalenti? Dobbiamo essere sicuri che, per ogni sequenza di ingresso $a_1 a_2 \cdots a_k$, lo stato S che l'automa D raggiunge quando

1. iniziamo dallo stato iniziale,
2. seguiamo il cammino etichettato con $a_1 a_2 \cdots a_k$,

è uno stato di accettazione se e solo se N accetta $a_1 a_2 \cdots a_k$. Si ricordi che N accetta $a_1 a_2 \cdots a_k$ se e solo se esiste almeno un cammino dallo stato iniziale di N, etichettato con $a_1 a_2 \cdots a_k$, che porta ad uno stato di accettazione di N.

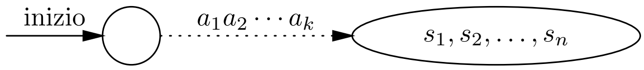
La relazione tra le azioni di D e quelle di N è ancora più forte. Se D ha un cammino dal suo stato iniziale allo stato S, etichettato da $a_1 a_2 \cdots a_k$, allora l'insieme S, visto come insieme di stati di N, è esattamente l'insieme degli stati raggiunti, a partire dallo stato iniziale di N, seguendo i cammini etichettati da $a_1 a_2 \cdots a_k$. Questa relazione è suggerita dalla Figura 10. Poiché abbiamo stabilito che S è uno stato di accettazione di D quando almeno uno degli elementi di S è uno stato di accettazione di N, la relazione mostrata dalla Figura 10 è esattamente quello che serve per concludere che D e N accettano entrambi o rifiutano entrambi $a_1 a_2 \cdots a_k$; cioè che D e N sono equivalenti.

Dobbiamo dimostrare la relazione di Figura 10; lo faremo per induzione su k, la lunghezza della stringa di ingresso. L'asserto formale da dimostrare per induzione su k è che lo stato $\{s_1, s_2, \ldots, s_n\}$ raggiunto in D seguendo il cammino etichettato con $a_1 a_2 \cdots a_k$ a partire dallo stato iniziale di D, è esattamente l'insieme degli stati di N che sono raggiunti dallo stato iniziale di N seguendo i cammini etichettati con $a_1 a_2 \cdots a_k$.

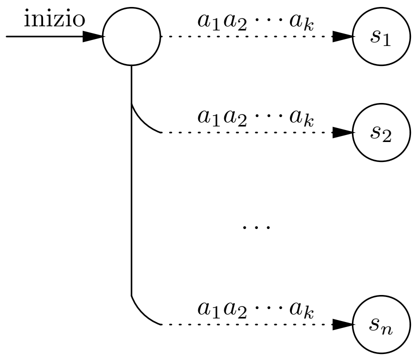
Base. Consideriamo $k = 0$. Un cammino di lunghezza 0 ci lascia nello stato di partenza, cioè nello stato iniziale di entrambi gli automi D e N. Si ricordi che, se s_0 è lo stato iniziale di N, allora lo stato iniziale di D è $\{s_0\}$. Quindi l'asserto vale per $k = 0$.

Induzione. Supponiamo che l'asserto valga per k e consideriamo la stringa di ingresso

$$a_1 a_2 \cdots a_k a_{k+1}$$



(a) Nell'automa D.



(b) Nell'automa N.

Figura 10: Relazione tra le azioni dell'automa non deterministico N e la sua controparte deterministica D.

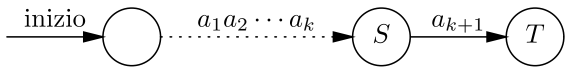


Figura 11: S è lo stato raggiunto da D immediatamente prima di raggiungere T

Allora il cammino dallo stato iniziale di D allo stato T, etichettato con $a_1a_2\cdots a_ka_{k+1}$ può essere visto come in Figura 11; cioè, passa attraverso uno stato S prima di fare l'ultima transizione a T con l'ingresso a_{k+1}.

Assumiamo, per ipotesi induttiva, che S sia esattamente l'insieme di stati che l'automa N raggiunge, a partire dal suo stato iniziale, attraverso i cammini etichettati con $a_1a_2\cdots a_k$; dobbiamo dimostrare che T è esattamente l'insieme degli stati che N raggiunge, a partire dal suo stato iniziale, attraverso i cammini etichettati con $a_1a_2\cdots a_ka_{k+1}$. La dimostrazione del passo induttivo è divisa in due parti.

1. Dobbiamo dimostrare che T non contiene troppi stati, cioè che, se t è uno stato di N che appartiene a T, allora t viene raggiunto attraverso un cammino etichettato con $a_1a_2\cdots a_ka_{k+1}$ a partire dallo stato iniziale di N.

2. Dobbiamo dimostrare che T contiene abbastanza stati, cioè che, se t è uno stato di N raggiunto attraverso un cammino etichettato con $a_1a_2\cdots a_ka_{k+1}$, a partire dallo stato iniziale, allora t appartiene a T.

Per quanto riguarda il punto (1), consideriamo uno stato t che appartiene a T. Allora, come suggerito dalla Figura 12, ci deve essere uno stato s in S che giustifichi il fatto che t appartiene a T. Cioè, c'è una transizione da s a t in N la cui etichetta contiene a_{k+1}. Per ipotesi induttiva, poiché s è in S, ci deve essere un cammino dallo stato iniziale di N a s, etichettato con $a_1a_2\cdots a_k$. Quindi c'è un cammino dallo stato iniziale di N a t etichettato con $a_1a_2\cdots a_ka_{k+1}$.

Adesso dobbiamo vedere il punto (2): se c'è un cammino dallo stato iniziale di N a t, etichettato con $a_1a_2\cdots a_ka_{k+1}$, allora t appartiene a T. Questo cammino deve passare attraverso uno stato s immediatamente prima di fare una transizione a t con ingresso a_{k+1}. Quindi c'è un cammino etichettato con $a_1a_2\cdots a_k$ dallo stato iniziale di N a s. Per ipotesi induttiva, s appartiene all'insieme di stati S. Poiché N ha una transizione da s a t, con una etichetta che include a_{k+1}, la costruzione dei sottoinsiemi applicata all'insieme di stati S e al simbolo di ingresso a_{k+1}, richiede che t sia inserito in T.

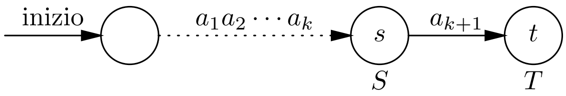


Figura 12: Lo stato s in S giustifica l'introduzione di t in T

Mediante l'ipotesi induttiva, abbiamo dimostrato che T è composto esattamente dagli stati di N che sono raggiungibili, dallo stato iniziale di N, attraverso i cammini etichettati da $a_1a_2\cdots a_ka_{k+1}$. Abbiamo quindi dimostrato il passo induttivo e possiamo concludere che lo stato dell'automa deterministico D, raggiunto attraverso il cammino etichettato con $a_1a_2\cdots a_k$, è sempre l'insieme degli stati di N raggiungibili attraverso i cammini con la stessa etichetta. Poiché gli stati di accettazione di D sono quelli che contengono uno stato di accettazione di N, possiamo concludere che D e N accettano le stesse stringhe; D e N sono dunque equivalenti e la costruzione dei sottoinsiemi "funziona correttamente".

Minimizzazione di automi

Una delle domande che ci si può porre riguardo agli automi è quanti stati siano necessari per riconoscere un certo linguaggio. Cioè ci possiamo chiedere, dato un automa, se esiste un automa equivalente con un numero minore di stati e, se esiste, qual è il più piccolo numero di stati di un automa equivalente.

Tra gli automi deterministici equivalenti ad uno dato, esiste un unico automa deterministico con il minimo numero di stati, detto *automa minimo*, ed è abbastanza semplice trovarlo. Il punto chiave è quello di determinare quando due stati s e t di un automa deterministico sono *equivalenti*, cioè, per ogni sequenza di ingresso, i cammini da s e da t etichettati con questa sequenza portano entrambi alla accettazione o portano entrambi alla non accettazione. Se gli stati s e t sono equivalenti, allora non esiste nessun ingresso che riesca a differenziarli e quindi possiamo fondere s e t in un singolo stato. In realtà è più facile definire quando due stati non sono equivalenti nel seguente modo.

Base. Se s è uno stato di accettazione e t non è uno stato di accettazione, o viceversa, allora s e t non sono equivalenti.

Induzione. Se esiste un simbolo di ingresso x tale che esistono transizioni da s e da t etichettate con x che portano a due stati non equivalenti, allora s e t non sono equivalenti.

È importante notare che non esiste una teoria della minimizazione per gli automi non deterministici.

Esercizi

2.3.1

Trasformare l'automa non deterministico dell'Esercizio 2.2.1 in un automa deterministico, usando la costruzione dei sottoinsiemi.

2.3.2

Quali stringhe riconoscono gli automi non deterministici di Figura 13, da (a) a (d)?

2.3.3

Trasformare gli automi non deterministici di Figura 13, da (a) a (d), in automi finiti deterministici.

2.3.4

L'automa deterministico di Figura 9 è minimo?

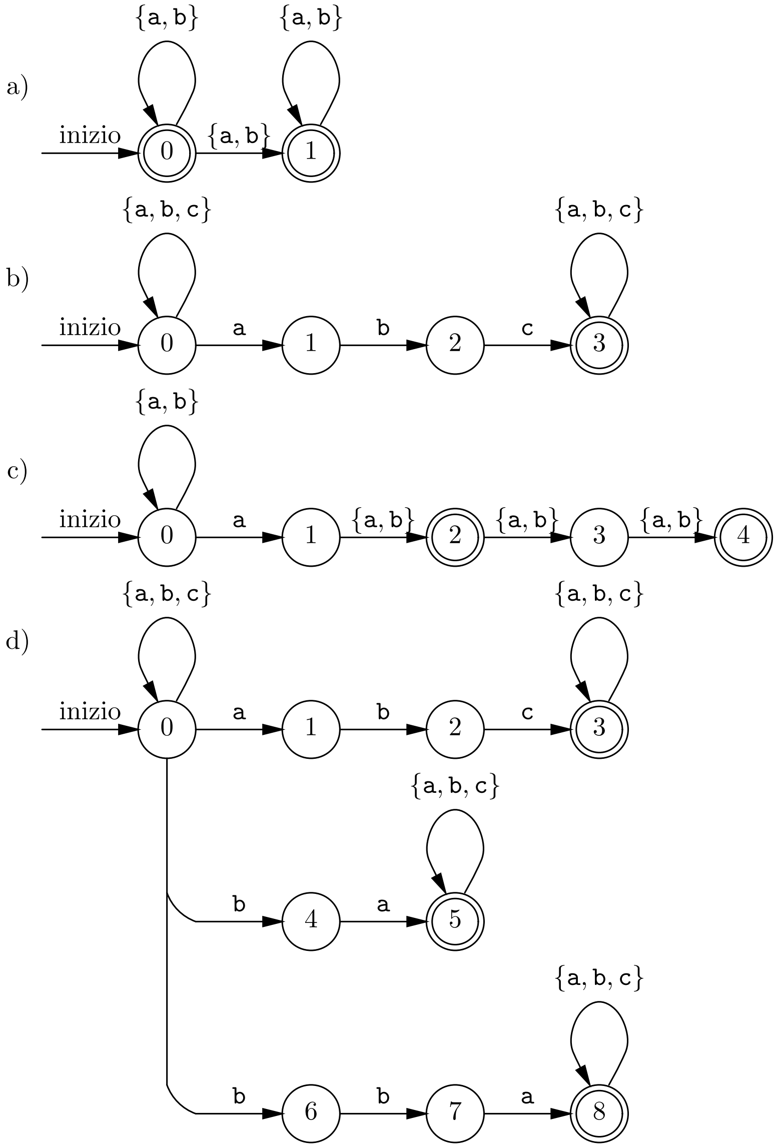


Figura 13: Automi non deterministici

2.4 Note Bibliografiche

Il lettore può trovare una trattazione più completa sugli automi e i linguaggi in Hopcroft e Ullman [1979].

Il modello degli automi per operare su stringhe, così come presentato in questo libro, è stato introdotto da Huffman [1954], sebbene ci siano stati modelli simili sviluppati prima o nello stesso tempo; la storia di questi modelli si trova in Hopcroft e Ullman [1979]. Gli automi non deterministici e la costruzione dei sottoinsiemi vengono da Rabin e Scott [1959].

Hopcroft, J. E., e J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, Massachusets, 1979.

Huffman, D. A.. "The synthesis of sequential switching machines". *Journal of the Franklin Institute* **257**:3-4, pp. 161–190 and 275–303, 1954.

Rabin, M. O. e D. Scott. "Finite automata and their decision problems", *IBM J. Research and Development* **3**:2, pp. 115-125, 1959.

3 Le Grammatiche

Nella sezione precedente abbiamo visto un modo per riconoscere le stringhe di un linguaggio che utilizza le etichette dei cammini di un grafo che abbiamo chiamato "automa". In questa sezione vedremo un altro modo per descrivere linguaggi, che fa uso di un tipo di definizione ricorsiva detta *grammatica libera da contesto* (per brevità, *grammatica*).

Un'importante applicazione delle grammatiche è la specifica della sintassi dei linguaggi di programmazione: le grammatiche costituiscono, infatti, una notazione concisa per descrivere la sintassi di un tipico linguaggio di programmazione e, in questa sezione, vedremo molti esempi a sostegno di questa affermazione. Vengono trattati i seguenti aspetti.

- Le grammatiche ed il loro uso nella definizione dei linguaggi (Paragrafi 3.1 e 3.2).
- Gli alberi di derivazione (*parse trees*), ovvero una rappresentazione ad albero della struttura delle stringhe rispetto ad una grammatica data (Paragrafo 3.3).
- L'ambiguità, cioè il problema che sorge quando una stringa possiede due o più alberi di derivazione diversi e dunque non possiede una "struttura" unica rispetto ad una grammatica data (Paragrafo 3.4).
- La dimostrazione che le grammatiche sono più potenti, per la descrizione dei linguaggi, degli automi (Paragrafo 3.5). Vedremo dapprima che le grammatiche sono espressive almeno quanto gli automi, mostrando come si possa simulare un automa mediante una grammatica. Successivamente, descriveremo un particolare linguaggio che può essere specificato da una grammatica ma non da un automa.

3.1 Grammatiche libere da contesto

Le espressioni aritmetiche possono essere definite ricorsivamente in modo naturale, come illustrato dall'esempio che segue. Consideriamo le espressioni aritmetiche che contengono:

1. i quattro operatori binari $+$, $-$, $*$ e $/$;
2. le parentesi;
3. i numeri come operandi.

Tali espressioni vengono di solito definite in modo induttivo come segue.

Base. Un numero è un'espressione.

Induzione. Se E è un'espressione, lo sono anche:

a) (E): possiamo cioè ottenere una nuova espressione racchiudendone un'altra tra parentesi;

b) $E + E$: due espressioni connesse dal segno "più" formano un'espressione;

c) $E - E$: questa e le due regole successive sono analoghe alla (b), ma usano gli altri operatori;

d) $E * E$;

e) E/E.

Questa induzione definisce un linguaggio, cioè un insieme di stringhe. La base afferma che ogni numero fa parte del linguaggio. La regola (a) afferma che, se s è una stringa del linguaggio, lo è anche la stringa (s): quest'ultima non è altro che s preceduta da una parentesi aperta e seguita da una parentesi chiusa. Le regole dalla (b) alla (e) affermano che, se s e t sono due stringhe del linguaggio, allora lo sono anche le stringhe s+t, s-t, s*t e s/t.

Le grammatiche consentono di scrivere queste regole in modo conciso e con un significato preciso. Come esempio, la nostra definizione delle espressioni aritmetiche potrebbe essere data mediante la grammatica di Figura 14. I simboli utilizzati nella Figura 14 necessitano di qualche spiegazione. Il simbolo

1. <*Espressione*> $\rightarrow$ <*Numero*>
2. <*Espressione*> $\rightarrow$ (<*Espressione*>)
3. <*Espressione*> $\rightarrow$ <*Espressione*> + <*Espressione*>
4. <*Espressione*> $\rightarrow$ <*Espressione*> − <*Espressione*>
5. <*Espressione*> $\rightarrow$ <*Espressione*> * <*Espressione*>
6. <*Espressione*> $\rightarrow$ <*Espressione*> / <*Espressione*>

Figura 14: Una grammatica per semplici espressioni aritmetiche

<*Espressione*> viene detto *categoria sintattica* e sta per una qualunque stringa nel linguaggio delle espressioni aritmetiche. Il simbolo $\rightarrow$ significa "può essere composto da": per esempio, la regola (2) di Figura 14 dice che un'espressione può essere composta da una parentesi aperta, seguita da una qualunque stringa che sia un'espressione, seguita da una parentesi chiusa. La regola (3) dice che un'espressione può essere composta da una qualunque stringa che sia un'espressione, seguita dal carattere `+`, seguito da una qualunque stringa che sia un'espressione. Le regole dalla (4) alla (6) sono simili alla regola (3). Nella regola (1) il simbolo <*Numero*> a destra della freccia è anch'esso una categoria sintattica, da interpretarsi come un segnaposto per una generica stringa che possa essere interpretata come un numero. Allo stato attuale non ci sono regole in cui <*Numero*> compaia a sinistra della freccia, e quindi non è ancora definito quali stringhe possano essere usate per rappresentare i numeri. Vedremo più avanti come si possano definire i numeri mediante una grammatica: per il momento immaginiamo che la grammatica delle espressioni utilizzi tale categoria sintattica per rappresentare gli operandi atomici.

Terminologia delle grammatiche

Nelle grammatiche si usano tre tipi di simboli: i primi sono i "metasimboli", ovvero simboli che ricoprono un ruolo speciale. L'unico esempio che abbiamo visto sin qui è il simbolo $\rightarrow$, che viene utilizzato per separare la categoria sintattica che si sta definendo dalla descrizione di uno dei possibili modi per formare le stringhe di quella categoria. Il secondo tipo di simbolo è la categoria sintattica, che, come detto, rappresenta gli insiemi di stringhe che stiamo definendo. Il terzo tipo di simbolo è il cosiddetto *simbolo terminale*: ovvero i simboli dell'alfabeto Λ a partire dai quali sono costruite le stringhe del linguaggio che si intende descrivere con una grammatica. Nell'esempio di Figura 14, i simboli terminali possono sono `+`, `-`, `*`, `/`, `)` e `(`.

Una grammatica è costituita da una o più *produzioni*: ogni linea della Figura 14 è una produzione. In generale, una produzione è formata da tre parti:

- una *testa,* che è la categoria sintattica a sinistra della freccia;
- il metasimbolo $\rightarrow$;
- il *corpo,* costituito da 0 o più categorie sintattiche e/o simboli terminali a destra della freccia.

Per esempio, nella regola (2) di Figura 14, la testa è <*Espressione*> ed il corpo è costituito da tre simboli: il simbolo terminale (, la categoria sintattica <*Espressione*> ed il simbolo terminale).

Convenzioni di notazione

Per denotare categorie sintattiche utilizziamo un nome, in corsivo, racchiuso tra parentesi angolate, come per esempio <*Espressione*>. Nelle produzioni, i simboli terminali sono denotati da **x**, in grassetto, che sta per la stringa `x`. Utilizziamo il metasimbolo ϵ per rappresentare un corpo vuoto: così, la produzione <*S*> $\rightarrow \epsilon$ significa che la stringa vuota fa parte del luinguaggio della categoria sintattica <*S*>. A volte raggruppiamo i corpi per una stessa categoria sintattica in un'unica produzione, separando i corpi con il metasimbolo |, che possiamo leggere come "oppure". Per esempio, se abbiamo le produzioni

$$<S> \to B_1, \quad <S> \to B_2, \quad \dots \quad , <S> \to B_n$$

in cui ogni B sta per il corpo di una produzione per la categoria sintattica <*S*>, possiamo scrivere le produzioni come:

$$<S> \to B_1 \ | \ B_2 \ | \ \cdots \ | \ B_n$$

Esempio 6 La definizione delle espressioni, con cui abbiamo iniziato questo paragrafo, può essere estesa fornendo una definizione per <*Numero*>. Supponiamo che i numeri siano stringhe di una o più cifre decimali. Tale idea può essere espressa nella notazione grammaticale, mediante le seguenti produzioni:

$<\mathit{Cifra}> \ \to \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$

$<\mathit{Numero}> \ \to \ <\mathit{Cifra}>$

$<\mathit{Numero}> \ \to \ <\mathit{Numero}> <\mathit{Cifra}>$

Si noti che, grazie alla nostra convenzione sull'uso del metasimbolo |, la prima linea è un'abbreviazione per le dieci produzioni

$<\mathit{Cifra}> \ \to \mathbf{0}$

$<\mathit{Cifra}> \ \to \mathbf{1}$

. . .

$<\mathit{Cifra}> \ \to \mathbf{9}$

In modo analogo, avremmo potuto combinare le produzioni per <*Numero*> in una sola linea. Osserviamo che la prima produzione per <*Numero*> dice che una singola cifra costituisce un numero, mentre la seconda produzione dice che anche un numero seguito da una cifra è un numero. Le due produzioni, insieme, dicono che un numero è una stringa formata da una o più cifre.

La Figura 15 presenta una grammatica estesa per le espressioni, completata con le produzioni per la categoria sintattica <*Numero*>. Si noti che la grammatica ha ora tre categorie sintattiche, <*Espressione*>, <*Numero*> e <*Cifra*>: tratteremo la categoria sintattica <*Espressione*> come il *simbolo iniziale*: esso genera le stringhe (in questo caso le espressioni aritmetiche ben formate) che intendiamo definire con la nostra grammatica. Le altre categorie sintattiche, <*Numero*> e <*Cifra*>, rappresentano dei concetti ausiliari, che sono essenziali ma che non costituiscono il concetto principale per cui è stata scritta la grammatica.

Esempio 7 Definiamo la categoria sintattica delle "stringhe di parentesi bilanciate" che potremmo chiamare <*Bilanciata*>. La regola di base dice che la stringa vuota è bilanciata: possiamo scrivere tale regola come la produzione

$<\mathit{Bilanciata}> \ \to \ \epsilon$

Nel passo induttivo possiamo dire che, se x e y sono stringhe bilanciate, lo è anche la stringa $(x)y$: possiamo ora scrivere questa regola come la produzione

$<\mathit{Bilanciata}> \ \to \mathbf{(} \ <\mathit{Bilanciata}> \ \mathbf{)} \ <\mathit{Bilanciata}>$

Dunque, possiamo dire che la grammatica composta dalle due produzioni precedenti definisce le stringhe di parentesi bilanciate.

Esiste un altro modo per descrivere le stringhe di parentesi bilanciate, legata all'idea che tali stringhe coincidono con le sottosequenze di parentesi che si ottengono cancellando, nelle espressioni, tutti i simboli eccetto le parentesi stesse. La Figura 14 descrive una grammatica per le espressioni: consideriamo ciò che succede cancellando tutti i simboli terminali, eccetto le parentesi. La produzione (1) diventa

1. $<$*Cifra*$>$ $\rightarrow$ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
2. $<$*Numero*$>$ $\rightarrow$ $<$*Cifra*$>$
3. $<$*Numero*$>$ $\rightarrow$ $<$*Numero*$>$ $<$*Cifra*$>$
4. $<$*Espressione*$>$ $\rightarrow$ $<$*Numero*$>$
5. $<$*Espressione*$>$ $\rightarrow$ ($<$*Espressione*$>$)
6. $<$*Espressione*$>$ $\rightarrow$ $<$*Espressione*$>$ + $<$*Espressione*$>$
7. $<$*Espressione*$>$ $\rightarrow$ $<$*Espressione*$>$ $-$ $<$*Espressione*$>$
8. $<$*Espressione*$>$ $\rightarrow$ $<$*Espressione*$>$ * $<$*Espressione*$>$
9. $<$*Espressione*$>$ $\rightarrow$ $<$*Espressione*$>$ / $<$*Espressione*$>$

Figura 15: La grammatica per le espressioni, con la definizione grammaticale dei numeri

$<$*Espressione*$>$ $\rightarrow$ ϵ

la produzione (2) diventa

$<$*Espressione*$>$ $\rightarrow$ ($<$*Espressione*$>$)

e le produzioni dalla (3) alla (6) diventano tutte

$<$*Espressione*$>$ $\rightarrow$ $<$*Espressione*$>$ $<$*Espressione*$>$

Se ora rimpiazziamo la categoria sintattica $<$*Espressione*$>$ con un nome più appropriato, $<$*BilanciataE*$>$, otteniamo un'altra grammatica per le stringhe di parentesi bilanciate, riportata in Figura 16. Queste produzioni sono piuttosto naturali, in quanto esprimono i seguenti fatti:

1. la stringa vuota è bilanciata;
2. racchiudendo tra parentesi una stringa bilanciata si ottiene ancora una stringa bilanciata;
3. concatenando due stringhe bilanciate si ottiene una stringa bilanciata.

$<$*BilanciataE*$>$ $\rightarrow$ ϵ

$<$*BilanciataE*$>$ $\rightarrow$ ($<$*BilanciataE*$>$)

$<$*BilanciataE*$>$ $\rightarrow$ $<$*BilanciataE*$>$ $<$*BilanciataE*$>$

Figura 16: Una grammatica per le stringhe di parentesi bilanciate.

Le due grammatiche per le stringhe di parentesi bilanciate proposte sembrano alquanto diverse, ma in realtà è possibile dimostrare che definiscono lo stesso insieme di stringhe.

Esempio 8 La struttura del controllo di un linguaggio di programmazione, come ad esempio il Pascal, può essere descritta in modo grammmaticale. Come semplice esempio, immaginiamo di avere già definito le produzioni per le categorie sintattiche $<$ *Condizione* $>$ e $<$ *ComSemplice* $>$. La prima è la categoria sintattica delle espressioni condizionali, ovvero delle espressioni a valori booleani. La categoria sintattica

$< \mathit{ComSemplice} >$ è quella dei comandi non strutturati, come un assegnamento, un comando di lettura e un comando di scrittura.

Useremo $<\mathit{Comando}>$ per la nostra categoria sintattica dei comandi Pascal. Un primo modo per costruire un comando fa uso del costrutto *while*: se abbiamo un comando che rappresenta il corpo del ciclo, possiamo farlo precedere dalla parola riservata **while**, da una condizione e dalla parola riservata **do**, ottenendo così un altro comando. La produzione per questa regola di formazione di comandi è la seguente:

$$<\mathit{Comando}> \;\rightarrow\; \textbf{while}\ < \mathit{Condizione} >\ \textbf{do}\ <\mathit{Comando}>$$

Un comando può anche essere costruito mediante il costrutto *if*: ci sono due forme diverse di comandi di questo tipo, a seconda della presenza o meno della parte *else*, espresse dalle seguenti regole:

$$<\mathit{Comando}> \;\rightarrow\; \textbf{if}\ < \mathit{Condizione} >\ \textbf{then}\ <\mathit{Comando}>$$

$$<\mathit{Comando}> \;\rightarrow\; \textbf{if}\ < \mathit{Condizione} >\ \textbf{then}\ <\mathit{Comando}>\ \textbf{else}\ <\mathit{Comando}>$$

Ci sono altri modi, in Pascal, per costruire comandi, come nel caso dei costrutti *for*, *repeat* e *case*: lasciamo per esercizio la scrittura delle produzioni corrispondenti, che sono simili a quelle appena viste.

Invece, un'altra regola importante per la formazione dei comandi, che in qualche modo si differenzia dalle regole analizzate fino ad ora, è costituita dal blocco: quest'ultimo utilizza le parole riservate **begin** e **end**, tra le quali è racchiusa una sequenza di uno o più comandi. Per descrivere i blocchi abbiamo bisogno di una categoria sintattica ausiliaria, che possiamo chiamare $<\mathit{ListaCom}>$, che rappresenti una sequenza di comandi. Le produzioni che definiscono $<\mathit{ListaCom}>$ sono semplici:

$$<\mathit{ListaCom}> \;\rightarrow\; <\mathit{Comando}>$$

$$<\mathit{ListaCom}> \;\rightarrow\; <\mathit{ListaCom}>;<\mathit{Comando}>$$

La prima produzione dice che qualunque comando è una sequenza unitaria di comandi; la seconda produzione dice che, se ad una sequenza di comandi facciamo seguire un punto e virgola ed un altro comando, otteniamo ancora una sequenza di comandi. Detto altrimenti, una $<\mathit{ListaCom}>$ è un $<\mathit{Comando}>$ seguito da zero o più coppie, ciascuna costituita da un punto e virgola e da un $<\mathit{Comando}>$.

A questo punto, i blocchi non sono altro che una sequenza di comandi racchiusa tra **begin** e **end**, cioè:

$$<\mathit{Comando}> \;\rightarrow\; \textbf{begin}\ <\mathit{ListaCom}>\ \textbf{end}$$

Le produzioni sin qui sviluppate, insieme con una produzione di base che dice che un comando può essere un comando semplice (assegnamento, chiamata, lettura o scrittura) sono riassunte nella Figura 17.

$$
\begin{array}{l}
<\mathit{Comando}> \;\rightarrow\; \textbf{while}\ < \mathit{Condizione} >\ \textbf{do}\ <\mathit{Comando}> \\
<\mathit{Comando}> \;\rightarrow\; \textbf{if}\ < \mathit{Condizione} >\ \textbf{then}\ <\mathit{Comando}> \\
<\mathit{Comando}> \;\rightarrow\; \textbf{if}\ < \mathit{Condizione} >\ \textbf{then}\ <\mathit{Comando}>\ \textbf{else}\ <\mathit{Comando}> \\
<\mathit{Comando}> \;\rightarrow\; \textbf{begin}\ <\mathit{ListaCom}>\ \textbf{end} \\
<\mathit{Comando}> \;\rightarrow\; < \mathit{ComSemplice} > \\
\\
<\mathit{ListaCom}> \;\rightarrow\; <\mathit{Comando}> \\
<\mathit{ListaCom}> \;\rightarrow\; <\mathit{ListaCom}>\ \textbf{;}\ <\mathit{Comando}>
\end{array}
$$

Figura 17: Le produzioni che definiscono alcuni comandi del Pascal

Formalmente una grammatica G è definita come una quadrupla

$$\langle \Lambda, V, S, P \rangle$$

dove

- Λ è un insieme di simboli detto alfabeto;
- V è l'insieme, finito, delle categorie sintattiche, ovvero di variabili che rappresentano sottolinguaggi;
- $S \in V$ è la categoria sintattica principale o iniziale;
- P è un insieme finito di produzioni. Ciascuna produzione, nel caso delle grammatiche libere ha la struttura

  $$A \to \alpha$$

  dove $A \in V$ e $\alpha \in (\Lambda \cup V)^+$.

Esercizi

3.1.1

Dare una grammatica per definire la categoria sintattica <*Identificatore*>, che corrispnde a tutte le stringhe che costituiscono gli identificatori in Pascal. Può essere d'aiuto definire alcune categorie sintattiche ausiliarie come <*Cifra*>.

3.1.2

In Pascal, gli operandi delle espressioni aritmetiche possono essere identificatori, oltre che numeri. Modificare la grammatica di Figura 15 in modo da consentire identificatori come operandi, utilizzando, per gli identificatori, la grammatica definita in risposta all'Esercizio 3.1.1.

3.1.3

Oltre agli interi, i numeri possono essere numeri reali, con il punto decimale ed un'eventuale potenza di 10. Modificare la grammatica per le espressioni di Figura 15, o la grammatica definita in risposta all'Esercizio 3.1.2, per consentire che gli operandi siano anche numeri reali.

3.1.4

A volte le espressioni possono contenere due o più tipi di parentesi bilanciate: per esempio, le espressioni in Pascal possono contenere sia parentesi tonde che parentesi quadre, ma entrambe devono essere bilanciate, nel senso che ad ogni parentesi `(` deve corrispondere una parentesi `)` e ad ogni parentesi `[` deve corrispondere una parentesi `]`. Scrivere una grammatica per le stringhe di parentesi bilanciate con entrambi i tipi di parentesi, che generi, cioè, tutte e sole le stringhe di parentesi che possono apparire in un'espressione Pascal ben formata.

3.1.5

Aggiungere alla grammatica di Figura 17 le produzioni per i comandi *for*, *repeat* e *case*: utilizzare in modo opportuno le categorie ausiliarie necessarie.

3.1.6

Espandere la grammatica nell'Esempio 8 completandola con la definizione della categoria sintattica <*Condizione*>, in modo da poter consentire l'uso degli operatori (connettivi) logici. Usare dapprima una categoria sintattica ausiliaria <*Confronto*> che esprima i confronti aritmetici in termini degli operatori di confronto come $<$ e della categoria sintattica <*Espressione*>, come ad esempio `x+1<y+z`. Quest'ultima può essere definita come all'inizio del Paragrafo 3.1, ma deve prevedere anche gli altri operatori Pascal, come il meno unario e `mod`.

3.1.7

Scrivere le produzioni che definiscono la categoria sintattica $<$*ComSemplice*$>$, in modo da rimpiazzare, nella Figura 17, la categoria sintattica $<$ *ComSemplice* $>$. Si può assumere che la categoria sintattica $<$*Espressione*$>$ corrisponda alle espressioni aritmetiche del Pascal. Ricordiamo che un "comando semplice" può essere un assegnamento, una chiamata di procedura, un comando di lettura o un comando di scrittura e che anche la stringa vuota rappresenta un comando semplice.

3.2 Linguaggi generati da grammatiche

Una grammatica è essenzialmente una definizione induttiva che coinvolge insiemi di stringhe: un fatto saliente è che spesso una stessa grammatica definisce varie categorie sintattiche contemporaneamente. Ad ogni categoria sintattica definita da una grammatica si può associare un linguaggio nel modo descritto di seguito.

Per ogni categoria sintattica $<S>$ di una grammatica, definiamo un linguaggio associato $L(<S>)$ nel modo seguente.

Passo Base. Si parte assumendo che, per ogni categoria sintattica $<S>$ della grammatica, il linguaggio $L(<S>)$ sia vuoto.

Passo Induttivo. Supponiamo che la grammatica contenga la produzione $<S> \rightarrow X_1 X_2 \cdots X_n$, in cui ogni X_i, per $i = 1, 2, \ldots, n$, è una categoria sintattica o un simbolo terminale. Per ciascun $i = 1, 2, \ldots, n$, si seleziona una stringa s_i per X_i nel modo seguente:

1. se X_i è un simbolo terminale, allora la stringa s_i è X_i stesso;
2. se X_i è una categoria sintattica, allora s_i è una qualunque stringa che già sappiamo appartenere a $L(X_i)$. Se la stessa categoria sintattica X_i appare più volte nel corpo della produzione, si può scegliere da $L(X_i)$ una stringa diversa per ogni occorrenza di X_i.

Allora, la concatenazione $s_1 s_2 \cdots s_n$ delle stringhe così selezionate appartiene al linguaggio $L(<S>)$: notiamo che, se $n = 0$, la stringa vuota ϵ appartiene al linguaggio.

Un modo sistematico per applicare questa definizione consiste nell'analizzare ciclicamente, una dopo l'altra, le produzioni della grammatica: si modifica ogni volta il linguaggio di ogni categoria sintattica usando la regola induttiva in tutti i modi possibili, facendo cioè tutte le scelte possibili per le stringhe s_i.

Esempio 9 Consideriamo la grammatica che contiene alcune produzioni della grammatica dell'Esempio 8 per i comandi Pascal. Per semplicità, useremo solo le produzioni per i comandi *while*, per il blocco e per i comandi semplici, oltre alle due produzioni per le sequenze di comandi. Inoltre, useremo un'abbreviazione che consente di abbreviare le stringhe in modo considerevole: faremo uso dei simboli terminali **w** (*while*), **c** (*condizione*), **d** (*do*), **b** (*begin*), **e** (*end*), **s** (al posto della categoria sintattica $<$ *ComSemplice* $>$) e punto e virgola. La grammatica utilizza la categoria sintattica $<S>$ per i comandi e la categoria sintattica $<L>$ per sequenze di comandi. Le produzioni sono mostrate in Figura 18.

1. $<S> \rightarrow \mathbf{w}\ \mathbf{c}\ \mathbf{d}\ <S>$
2. $<S> \rightarrow \mathbf{b}\ <L>\ \mathbf{e}$
3. $<S> \rightarrow \mathbf{s}$
4. $<L> \rightarrow <L>\ \mathbf{;}\ <S>$
5. $<L> \rightarrow <S>$

Figura 18: La grammatica dei comandi semplificata

Siano L il linguaggio di stringhe della categoria sintattica $<L>$ e S quello delle stringhe della categoria sintattica $<S>$. Inizialmente, per la regola base, sia L che S sono vuoti. Al primo ciclo di analisi, solo la produzione (3) è di qualche utilità, poiché il corpo di tutte le altre produzioni contiene una categoria sintattica ed ancora non abbiamo nessuna stringa che appartenga al linguaggio di tale categoria. La produzione (3) ci consente di inferire che la stringa `s` appartiene al linguaggio S.

Iniziamo il secondo ciclo con L vuoto e $S = \{\mathtt{s}\}$. La produzione (1) ci consente ora di aggiungere `wcds` a S, poiché `s` appartiene già a S. Nel corpo della produzione (1), i simboli terminali **w**, **c** e **d** possono rappresentare solo se stessi, ma la categoria sintattica $<S>$ può essere rimpiazzata da una qualunque stringa del linguaggio S: essendo, al momento, la stringa s l'unico elemento di S, possiamo fare una sola scelta che produce la stringa `wcds`.

La produzione (2) non aggiunge alcunché, visto che L è ancora vuoto. Poiché il corpo della produzione (3) contiene solo un simbolo terminale, tale produzione non produrrà mai altre stringhe se non `s` e quindi possiamo, di qui in poi, ignorarla. La produzione (4) ancora non può essere utilizzata, ma possiamo usare la produzione (5) per aggiungere `s` a L. Alla fine di questo ciclo di analisi delle produzioni, abbiamo ottenuto i linguaggi $S = \{\mathtt{s}, \mathtt{wcds}\}$ e $L = \{\mathtt{s}\}$.

Al ciclo successivo possiamo usare, per generare nuove stringhe, le produzioni (1), (2), (4) e (5). Nella produzione (1) abbiamo due modi possibili per sostituire $<S>$, vale a dire `s` e `wcds`. Nel primo caso otteniamo, per il linguaggio S, una stringa che già conosciamo, ma nel secondo caso otteniamo la nuova stringa `wcdwcds`. La produzione (2) ci consente solo di sostituire $<L>$ con `s`, ottenendo la stringa `bse` del linguaggio S. Nella produzione (4), possiamo sostituire $<L>$ solo con `s`, ma abbiamo due scelte possibili per $<S>$, vale a dire `s` o `wcds`, ottenendo così le due stringhe `s;s` e `s;wcds` per il linguaggio L. Infine, la produzione (5) dà luogo alla nuova stringa `wcds` per il linguaggio L. Stiamo procedendo in modo molto sistematico nella sostituzione di categorie sintattiche con stringhe. Si noti che stiamo supponendo che, ad ogni ciclo, i linguaggi L e S siano fissati e siano definiti come al termine del ciclo precedente; le sostituzioni vengono effettuate in tutti i corpi delle produzioni. Consentiamo che i corpi delle produzioni producano nuove stringhe per le teste corrispondenti, ma, nell'analisi di una produzione, non consentiamo l'utilizzo delle nuove stringhe generate da un altro corpo all'interno dello stesso ciclo. Ciò non costituisce un limite: tutte le stringhe che devono essere generate verrano prima o poi generate, indipendentemente dal fatto che consentiamo l'utilizzo immediato delle stringhe appena prodotte o ne consentiamo l'uso al ciclo successivo.

I linguaggi che abbiamo a questo punto sono $S = \{\mathtt{s}, \mathtt{wcds}, \mathtt{wcdwcds}, \mathtt{bse}\}$ e

$$L = \{\mathtt{s}, \mathtt{s;s}, \mathtt{s;wcds}, \mathtt{wcds}\}.$$

Possiamo procedere quanto vogliamo in questo modo: la Figura 19 riassume il risultato dei primi tre cicli.

	S	L
Ciclo 1:	`s`	
Ciclo 2:	`wcds`	`s`
Ciclo 3:	`wcdwcds`	`s;wcds`
	`bse`	`s;s`
		`wcds`

Figura 19: Le stringhe generate nei primi tre cicli

Il linguaggio generato da una grammatica è in generale infinito. Se un linguaggio è infinito, non possiamo elencare tutte le sue stringhe: il meglio che possiamo fare è *enumerare* le stringhe ad ogni ciclo, come avevamo cominciato a fare nell'Esempio 9. Ogni stringa del linguaggio, apparirà al termine di qualche ciclo, ma non può mai succedere che alla fine di un ciclo siano state prodotte tutte le stringhe. L'insieme delle stringhe che vengono a far parte, prima o poi, del linguaggio della categoria sintattica $<S>$ costituisce il linguaggio (infinito) $L(<S>)$.

Esercizi

3.2.1

Quali nuove stringhe vengono aggiunte, nell'Esempio 9, al termine del quarto ciclo?

3.2.2

Utilizzando le due diverse grammatiche proposte nell'esempio 7, generare ciclicamente le stringhe di parentesi bilanciate. È vero che, ad uno stesso ciclo, le due grammatiche generano lo stesso insieme di stringhe?

3.2.3

Supponiamo che ogni produzione la cui testa è $<S>$ contenga $<S>$ anche nel corpo. Perché in questo caso $L(<S>)$ è vuoto?

3.2.4

Quando si generano le stringhe per cicli successivi, come descritto in questo paragrafo, per ogni categoria sintattica $<S>$, le nuove stringhe vengono prodotte sostituendo, nel corpo di qualche produzione per $<S>$, le categorie sintattiche con stringhe, in modo che almeno una di queste ultime sia stata prodotta al ciclo precedente. Spiegare perché questa affermazione è vera.

3.2.5

Supponiamo di voler conoscere se una stringa s appartenga o meno al linguaggio di una categoria sintattica $<S>$.

a) Spiegare perché, se al termine di un ciclo tutte le nuove stringhe generate per qualunque categoria sintattica sono più lunghe di s, e s non è ancora stata generata in $L(<S>)$, allora s non farà mai parte di $L(<S>)$.

b) Spiegare perché, dopo un numero finito di cicli, non possiamo più generare stringhe la cui lunghezza sia minore o uguale della lunghezza di s.

c) Utilizzare (a) e (b) per sviluppare un algoritmo che, date una grammatica, una delle sue categorie sintattiche $<S>$ ed una stringa di simboli terminali s, dica se s fa o meno parte di $L(<S>)$.

3.3 Alberi di analisi

Come abbiamo visto, è possibile scoprire se una stringa s appartiene al linguaggio $L(<S>)$, per una categoria sintattica $<S>$, applicando ripetutamente le produzioni della grammatica. Partiamo da alcune stringhe derivate mediante le produzioni di base, quelle cioè che non contengono categorie sintattiche nel corpo; quindi, "applichiamo" le produzioni alle stringhe già derivate per le varie categorie sintattiche. Ogni applicazione richiede la sostituzione delle varie categorie sintattiche nel corpo di una produzione, portando così a formare una stringa che appartiene alla categoria sintattica della testa. In questo modo, la stringa verrà prima o poi ottenuta applicando una produzione con testa $<S>$.

Spesso è utile rappresentare la "dimostrazione" che s appartiene a $L(<S>)$ mediante una struttura detta *albero*.

Un albero è un tipo particolare di grafo, caratterizzato dai seguenti due vincoli:

1. non esistono cicli (grafo aciclico) e
2. ogni nodo del grafo ha al più un arco entrante.

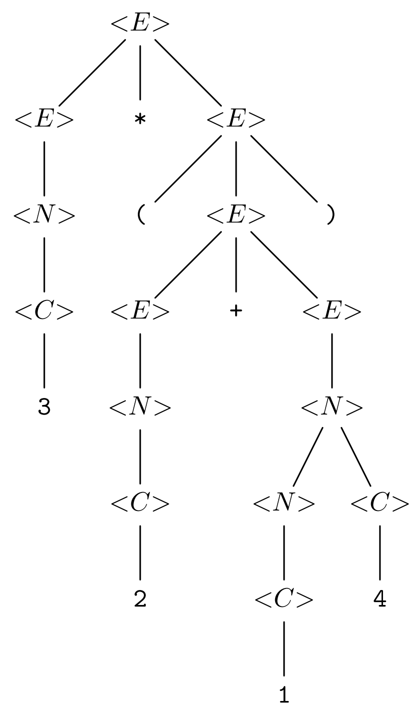


Figura 20: L'albero di analisi della stringa $\mathtt{3} * (\mathtt{2} + \mathtt{14})$, usando la grammatica della Figura 15

Ad esempio, il grafo di Figura 6, *non* è un albero per due motivi distinti: esitono cicli (ad esempio il nodo 0 ha un arco ciclico), ed esistono nodi, come il nodo 3, con più di un arco entrante. La Figura 20 mostra invece un esempio di albero. Nella figura, gli archi devono essere pensati come orientati verso il basso: la freccia è omessa per non appesantire il disegno. Per lo stesso motivo, sono omessi nella figura i cerchi che contraddistinguono i nodi. Ogni nodo nel grafo di Figura 20 ha esattamente un arco entrante, ad eccezione del nodo più in alto, etichettato con $< E >$. Tale nodo è detto *radice* dell'albero. I nodi raggiungibili tramite gli archi uscenti da un dato nodo v di un albero sono detti i *figli* di v. Nella Figura 20, la radice dell'albero ha tre figli, etichettati con $< E >$, $*$ e $< E >$. Viceversa, per ogni nodo v di un albero tranne la radice esiste un unico *genitore* del nodo v, che consiste nel nodo da cui esce l'unico arco entrante in v. Proseguendo con la terminologia di carattere "botanico", i nodi che non hanno figli (ovvero, che non hanno archi uscenti) sono detti *foglie* dell'albero. Ad esempio, il nodo etichettato con 3 nell'albero di Figura 20 è una foglia. Viceversa, i nodi che hanno almeno un figlio (ovvero, almeno un arco uscente) sono detti *nodi interni* dell'albero. Infine, la stringa che si ottiene concatenando le etichette delle foglie di un albero da sinistra verso destra è detta la *frontiera* dell'albero. La frontiera dell'albero di Figura 20 consiste nella stringa $\mathtt{3} * (\mathtt{2} + \mathtt{14})$.

La struttura ad albero è particolarmente adatta a rappresentare la "dimostrazione" che una stringa appartiene al linguaggio generato da una grammatica: un tale albero è chiamato *albero di derivazione* (in inglese *parse tree*). I nodi di un albero di derivazione sono etichettati da simboli terminali, oppure da categorie sintattiche, oppure dal simbolo ϵ: le foglie sono etichettate solo da simboli terminali o da ϵ, mentre i nodi interni sono etichettati solo da categorie sintattiche.

Ogni nodo interno v rappresenta l'applicazione di una produzione, ovvero deve esistere una produzione tale che:

1. la categoria sintattiche che etichetta v sia la testa della produzione;

2. le etichette dei figli di v, da sinistra a destra, costituiscano il corpo della produzione.

Esempio 10 La Figura 20 è un esempio di albero di derivazione, basato sulla grammatica di Figura

15, in cui abbiamo però abbreviato le categorie sintattiche $<Espressione>$, $<Numero>$ e $<Cifra>$ rispettivamente con $<E>$, $<N>$ e $<C>$. La stringa rappresentata dall'albero di derivazione è `3*(2+14)`.

Per esempio, la radice ed i suoi figli rappresentano la produzione

$$<E> \rightarrow <E> \texttt{*} <E>$$

ovvero la produzione (6) di Figura 15. Il figlio più a destra della radice, con i suoi tre figli, rappresenta la produzione

$$<E> \rightarrow \texttt{(}<E>\texttt{)}$$

ovvero la produzione (5) di Figura 15.

Costruzione degli alberi di derivazione

Ogni albero di derivazione rappresenta una stringa di simboli terminali s, che chiamiamo il *prodotto* dell'albero. La stringa s è costituita dalla frontiera dell'albero, ovvero dalle etichette delle foglie dell'albero, prese da sinistra a destra. Si ricordi che le foglie (e solo le foglie) di un albero di derivazione sono etichettate da simboli terminali. Per esempio, il prodotto dell'albero di derivazione di Figura 20 è `3*(2+14)`.

Se un albero ha un solo nodo, allora quel nodo, essendo una foglia, sarà etichettato da un simbolo terminale o da ϵ. Se un albero, invece, ha più di un nodo, allora la radice sarà etichettata da una categoria sintattica, dal momento che la radice di un albero con due o più nodi è comunque un nodo interno: tale categoria sintattica conterrà sempre, tra le sue stringhe, il prodotto dell'albero. Quella che segue è una definizione induttiva di albero di derivazione per una grammatica data.

Passo Base. Per ogni simbolo terminale della grammatica, sia esso `x`, c'è un albero con un solo nodo etichettato con `x`. Il prodotto di tale albero, naturalmente, è `x`.

Passo Induttivo. Supponiamo di avere una produzione $<S> \to X_1X_2\cdots X_n$, in cui ogni X_i è un simbolo terminale o una categoria sintattica. Se $n = 0$, se cioè la produzione è, in realtà, $<S> \to \epsilon$, allora c'è un albero come quello di Figura 21. Il prodotto è ϵ e la radice è $<S>$; sicuramente la stringa ϵ appartiene $L(<S>)$, grazie a questa produzione.

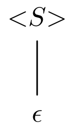


Figura 21: Albero di derivazione per la produzione $<S> \to \epsilon$

Supponiamo ora che $<S> \to X_1X_2\cdots X_n$ e $n \geq 1$. Possiamo scegliere un albero T_i per ogni X_i, $i = 1, 2, \ldots, n$, come segue.

1. Se X_i è un simbolo terminale, dobbiamo scegliere l'albero con un solo nodo etichettato da X_i. Se due o più X sono lo stesso simbolo terminale, allora dobbiamo scegliere un albero diverso, con un solo nodo, per ogni occorrenza di questo simbolo terminale.

2. Se X_i è una categoria sintattica, possiamo scegliere un albero di derivazione già costruito e la cui radice sia etichettata con X_i. Costruiamo così un albero simile a quello di Figura 22: creiamo, cioè, una radice etichettata con $<S>$, la categoria sintattica che appare nella testa della produzione, e diamo ad essa, come figli, gli alberi che abbiamo selezionato per $X_1, X_2, \ldots, X_n$, nell'ordine da sinistra a destra. Se due o più X sono la stessa categoria sintattica, possiamo scegliere per ciascuna lo stesso albero, ma, ogni volta che lo selezioniamo, dobbiamo farne una copia diversa. Possiamo altresì scegliere alberi diversi per occorrenze diverse della stessa categoria sintattica.

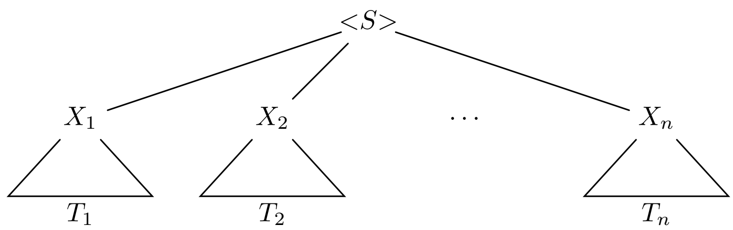
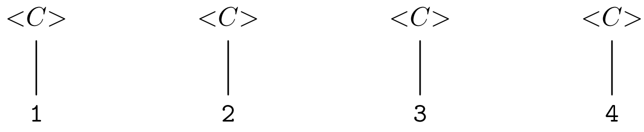


Figura 22: Costruzione di un albero di derivazione usando una produzione ed altri alberi

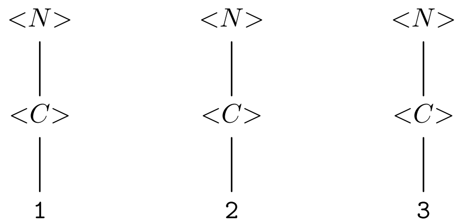
Esempio 11 Seguiamo la costruzione dell'albero di derivazione di Figura 20 e vediamo come tale costruzione rispecchi la dimostrazione che la stringa `3*(2+14)` appartiene a $L(<E>)$. Innanzitutto, possiamo costruire un albero di un solo nodo per ciascuno dei simboli terminali dell'albero. Il gruppo di produzioni alla linea (1) di Figura 15 afferma che ciascuna delle dieci cifre è una stringa, di lunghezza unitaria, appartenente a $L(<C>)$. Usiamo quattro di queste produzioni per creare i quattro alberi di Figura 23. Per esempio, l'albero di Figura 23(a) viene costruito, a partire dalla produzione $<C> \rightarrow \mathbf{1}$, nel modo seguente. Creiamo un singolo nodo etichettato con `1` corrispondente al simbolo **1** del corpo della produzione; quindi creiamo un nodo etichettato con $<C>$, come radice, e diamo ad esso come figlio la radice (ed unico nodo) dell'albero selezionato per **1**.



(a) (b) (c) (d)

Figura 23: Alberi di derivazione costruiti usando la produzione $<C> \rightarrow \mathbf{1}$ ed altre simili.

Il nostro passo successivo consiste nell'utilizzare la produzione (2) di Figura 15, ovvero $<N> \rightarrow <C>$, per scoprire che le cifre sono numeri. Per esempio, possiamo scegliere l'albero di Figura 23(a) da sostituire a $<C>$ nel corpo della produzione (2) ed ottenere così l'albero di Figura 24(a); gli altri due alberi di Figura 24 vengono costruiti in modo analogo.



(a) (b) (c)

Figura 24: Alberi di derivazione costruiti utilizzando la produzione $<N> \rightarrow <C>$

Possiamo ora utilizzare la produzione (3), che è $<N> \rightarrow <N><C>$. Per $<N>$ nel corpo possiamo scegliere l'albero di Figura 24(a) e per $<C>$ l'albero di Figura 23(d): creiamo un nuovo nodo etichettato con $<N>$, la testa della produzione, e diamo ad esso due figli, che sono le radici dei due alberi selezionati. L'albero risultante è mostrato in Figura 25: il suo prodotto è il numero `14`.

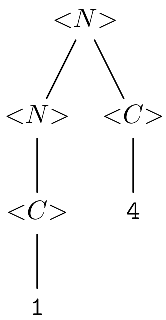


Figura 25: Alberi di derivazione costruiti utilizzando la produzione $<N> \rightarrow <N><C>$

Il nostro compito successivo è di costruire un albero per la somma `2+14`. In primo luogo, utilizziamo la produzione (4), ovvero $<E> \rightarrow <N>$, in modo da costruire gli alberi di Figura 26. Questi ultimi mostrano che `3`, `2` e `14` sono espressioni. Il primo di questi alberi risulta dalla scelta dell'albero di Figura 24(c) per la categoria $<N>$ del corpo; il secondo risulta dalla scelta, per $<N>$, dell'albero di Figura 24(b) ed il terzo dalla scelta dell'albero di Figura 25.

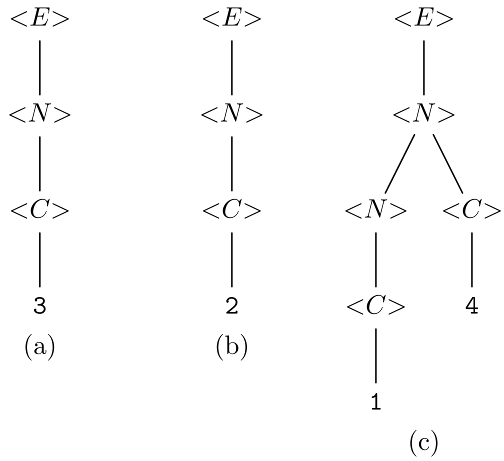


Figura 26: Alberi di derivazione costruiti utilizzando la produzione $<E> \rightarrow <N>$

Utilizziamo poi la produzione (6), che è $<E> \rightarrow <E>$+$<E>$. Per la prima occorrenza di $<E>$ nel corpo usiamo l'albero di Figura 26(b) e per la seconda occorrenza di $<E>$ usiamo l'albero di Figura 26(c). Per il simbolo terminale + nel corpo, usiamo l'albero con un solo nodo etichettato con `+`. L'albero risultante è mostrato in Figura 27: il suo prodotto è `2+14`.

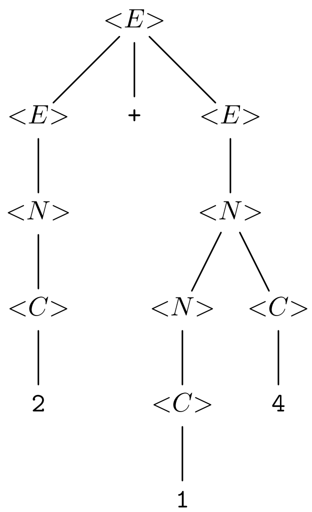


Figura 27: Albero di derivazione costruito utilizzando la produzione $<E> \rightarrow <E>$+$<E>$

Utilizziamo quindi la produzione (5), ovvero $<E> \rightarrow$ ($<E>$), per costruire l'albero di derivazione di Figura 28: abbiamo semplicemente scelto l'albero di Figura 27 per l'occorrenza di $<E>$ nel corpo e, ovviamente, gli alberi con un solo nodo per le sue parentesi.

Infine, usiamo la produzione (8), che è $<E> \rightarrow <E>$ `*` $<E>$, per costruire l'albero di derivazione che abbiamo mostrato inizialmente in Figura 20. Per la prima occorrenza di $<E>$ nel corpo, scegliamo l'albero di Figura 26(a) e per la seconda scegliamo l'albero di Figura 28.

Perché gli alberi di derivazione "funzionano"

La costruzione degli alberi di derivazione è molto simile alla definizione induttiva delle stringhe che appartengono ad una categoria sintattica. È possibile dimostrare, mediante due semplici induzioni, che i prodotti degli alberi di derivazione con radice $<S>$ sono esattamente le stringhe appartenenti a $L(<S>)$, per ogni categoria sintattica $<S>$. Più precisamente:

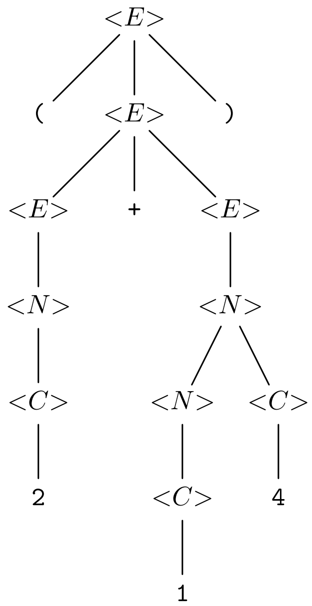


Figura 28: Albero di derivazione costruito utilizzando la produzione $<E> \rightarrow$ `(`$<E>$`)`

1. se T è un albero di derivazione la cui radice è etichettata con $<S>$ ed il cui prodotto è s, allora la stringa s appartiene al linguaggio $L(<S>)$;
2. se la stringa s appartiene a $L(<S>)$, allora esiste un albero di derivazione il cui prodotto è s e la cui radice è etichettata con $<S>$.

La dimostrazione di questo risultato è omessa in queste note.

Esercizi

3.3.1

Determinare l'albero di derivazione per le seguenti espressioni:

a) `35+21`

b) `123-(4*5)`

c) `1*2*(3-4)`

in riferimento alla grammatica di Figura 15. In ciascun caso, l'etichetta del nodo radice deve essere $<E>$.

3.3.2

Utilizzando la grammatica per i comandi di Figura 18, determinare gli alberi di derivazione per le seguenti stringhe:

a) `wcdwcds`

b) `bse`

c) `bs;wcdse`.

In ciascun caso, l'etichetta del nodo radice deve essere $<S>$.

3.3.3

Usando le grammatiche per le parentesi bilanciate dell'Esempio 7, determinare gli alberi di derivazione per le seguenti stringhe:

a) `(()())`

b) `((()))`

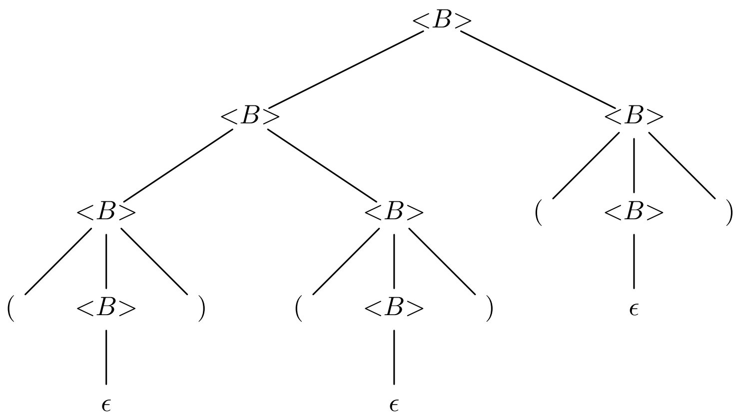
c) `((())()).`

3.4 Ambiguità e progetto di grammatiche

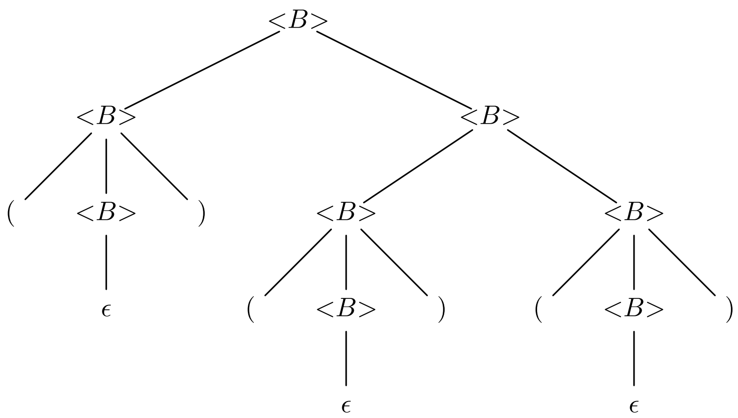
Consideriamo la grammatica per le parentesi bilanciate, mostrata originariamente in Figura 16, in cui la categoria sintattica $<B>$ viene usata come abbreviazione di $<\mathit{BilanciataE}>$:

$$<B> \to \texttt{(}<B>\texttt{)} \mid <B><B> \mid \epsilon \tag{1}$$

Supponiamo di voler costruire un albero di derivazione per la stringa `()()()`; la Figura 29 mostra due alberi di derivazione: il primo corrisponde a raggruppare dapprima le due parentesi iniziali, mentre il secondo corrisponde a raggruppare dapprima le due parentesi finali.



(a) Albero di derivazione che raggruppa da sinistra.



(b) Albero di derivazione che raggruppa da destra.

Figura 29: Due alberi di derivazione con la stessa radice e lo stesso prodotto

Non dovrebbe sorprendere il fatto che esistano due alberi siffatti: una volta stabilito che `()` e `()()` sono entrambe stringhe di parentesi bilanciate, possiamo usare la produzione $<B> \to <B><B>$ sostituendo la prima e la seconda occorrenza di $<B>$ nel corpo rispettivamente con con `()` e `()()`, o viceversa. In entrambi i modi si scopre che la stringa `()()()` appartiene al linguaggio della categoria sintattica $<B>$.

Una grammatica in cui ci sono due o più alberi di derivazione con lo stesso prodotto e con la radice etichettata dalla stessa categoria sintattica si dice *ambigua*. Si noti che non è necessario che ogni stringa sia il prodotto di più alberi di derivazione: affinché la grammatica sia ambigua è sufficiente che ci sia anche soltanto una stringa siffatta. Per esempio, la stringa `()()()` è sufficiente a concludere che la grammatica (1) è ambigua. Una grammatica che non sia ambigua viene detta, appunto, *non ambigua*. In una grammatica non ambigua, per ogni stringa s e categoria sintattica $<S>$, esiste al più un albero di derivazione il cui prodotto è s e la cui radice è etichettata con $<S>$.

Per esempio, l'altra grammatica per le parentesi bilanciate dell'Esempio 7, che riportiamo qui di seguito con $<B>$ al posto di $<\mathit{Bilanciata}>$,

$$<B> \to (<B>)<B> \mid \epsilon \qquad (2)$$

è una grammatica non ambigua. È piuttosto difficile dimostrare che una grammatica non è ambigua. Nella Figura 30 è riportato l'unico albero di derivazione per la stringa `()()()`: il fatto che tale stringa abbia un solo albero di derivazione non basta per dimostrare che la grammatica (2) non è ambigua, poiché la non ambiguità può essere dimostrata soltanto mostrando che *ogni* stringa del linguaggio possiede un unico albero di derivazione.

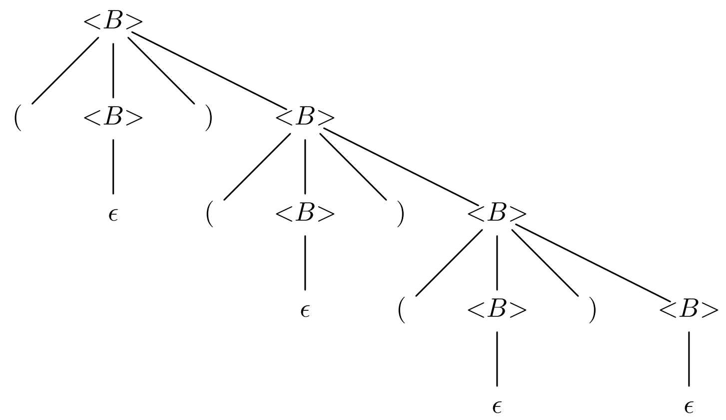


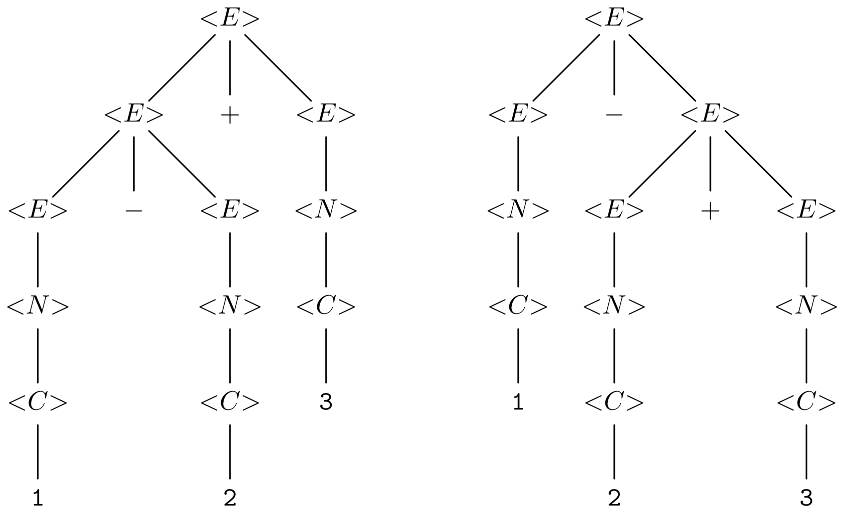
Figura 30: L'unico albero di derivazione per la stringa `( ) ( ) ( )` usando la grammatica (2)

Ambiguità nelle espressioni

Benché la grammatica di Figura 16 sia ambigua, tale ambiguità non è molto dannosa, dal momento che non fa differenza raggruppare le stringhe di parentesi bilanciate da sinistra o da destra. Quando però consideriamo le grammatiche per le espressioni, come quella della Figura 15 nel Paragrafo 3.1, possono presentarsi problemi più seri. In particolare, alcuni alberi di derivazione corrispondono ad una valutazione errata dell'espressione, mentre altri corrispondono alla valutazione corretta.

Esempio 12 Utilizziamo la notazione abbreviata per la grammatica delle espressioni che abbiamo sviluppato nell'Esempio 10; consideriamo poi l'espressione `1-2+3`, a cui corrispondono due alberi di derivazione, mostrati in Figura 31(a) e (b), a seconda che si raggruppino gli operandi da sinistra o da destra.

L'albero di Figura 31(a) raggruppa gli operandi da sinistra e ciò è corretto, poiché di solito gli operatori con la stessa precedenza vengono raggruppati da sinistra: convenzionalmente, `1-2+3` viene interpretata come `(1-2)+3`, il cui valore è 2. Se, analizzando l'albero di Figura 31(a) dal basso verso l'alto, valutiamo le espressioni rappresentate dai sottoalberi, prima calcoliamo, in corrispondenza del figlio sinistro della radice, $1 - 2 = -1$ e poi, in corrispondenza della radice, calcoliamo $-1 + 3 = 2$.



(a) Albero di derivazione corretto. (b) Albero di derivazione scorretto.

Figura 31: Due alberi di derivazione per l'espressione `1` $-$ `2` $+$ `3`

D'altra parte, nella Figura 31(b), che corrisponde a raggruppare da destra, la nostra espressione viene interpretata come `1-(2+3)`, il cui valore è -4: questa però non è l'interpretazione convenzionale. Il valore -4 si ottiene analizzando l'albero di Figura 31(b) dal basso verso l'alto e valutando prima $2+3=5$, in corrispondenza del figlio destro della radice, e poi $1-5=-4$, in corrispondenza della radice.

Raggruppando gli operatori con la stessa precedenza nella direzione sbagliata si possono creare problemi, così come raggruppando un operatore con minore precedenza prima di uno con maggiore precedenza.

Esempio 13 Consideriamo l'espressione `1+2*3`: in Figura 32(a) abbiamo raggruppato l'espressione, scorrettamente, da sinistra, mentre in Figura 32(b) l'abbiamo raggruppata correttamente da destra, cosicché gli operandi della moltiplicazione vengono raggruppati prima dell'addizione. Nel primo caso otteniamo il valore errato 9, mentre nel secondo otteniamo il valore convenzionale 7.

Grammatiche non ambigue per le espressioni

Così come la grammatica (2) per le parentesi bilanciate può essere vista come la versione non ambigua della grammatica (1), è possibile costruire una versione non ambigua della grammatica per le espressioni dell'Esempio 10. Il "trucco" sta nel definire tre categorie sintattiche il cui significato intuitivo è il seguente:

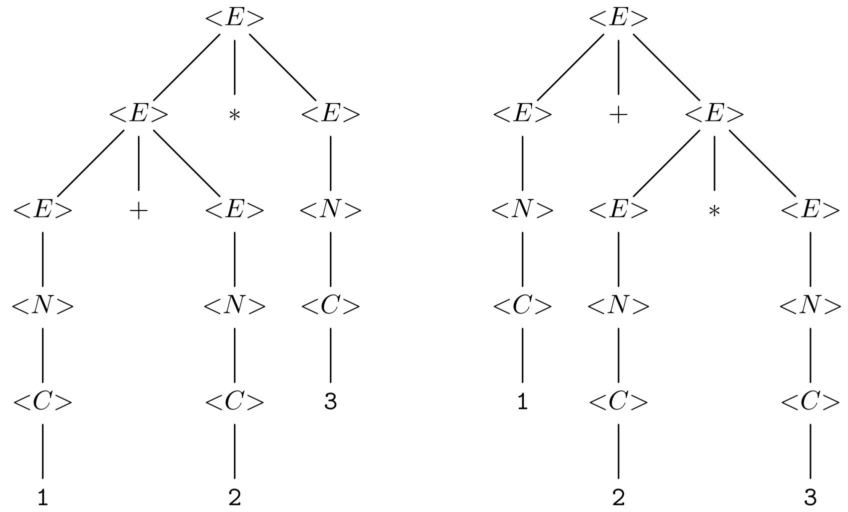
1. $<$*Fattore*$>$ genera le espressioni che non si possono "spezzare": un fattore, cioè, è un singolo operando o una espressione tra parentesi;

2. $<$*Termine*$>$ genera un prodotto o un quoziente di fattori: un singolo fattore è un termine e tale è anche una sequenza di fattori separati dagli operatori $*$ o / (esempi di termini sono `12` e `12/3*45`);

3. $<$*Espressione*$>$ genera la somma o la differenza di uno o più termini: un singolo termine è un'espressione e tale è anche una sequenza di termini separati dall'operatore $+$ o $-$ (esempi di espressioni sono `12`, `12/3*45` e `12+3*45-6`).

Nella Figura 33 è mostrata una grammatica che fornisce le relazioni tra espressioni, termini e fattori, in cui abbiamo utilizzato le abbreviazioni $<E>$, $<T>$ e $<F>$ per, rispettivamente, $<$*Espressione*$>$, $<$*Termine*$>$ e $<$*Fattore*$>$.

Per esempio, le tre produzioni alla linea (1) definiscono un'espressione come un'espressione seguita da `+` o `-` e da un altro termine, oppure come un singolo termine. Mettendole insieme, queste produzioni



(a) Albero di derivazione scorretto. (b) Albero di derivazione corretto.

Figura 32: Due alberi di derivazione per l'espressione `1+2*3`

1. $<\!E\!> \rightarrow <\!E\!> + <\!T\!> \mid <\!E\!> - <\!T\!> \mid <\!T\!>$
2. $<\!T\!> \rightarrow <\!T\!> * <\!F\!> \mid <\!T\!>/<\!F\!> \mid <\!F\!>$
3. $<\!F\!> \rightarrow \textbf{(}<\!E\!>\textbf{)} \mid <\!N\!>$
4. $<\!N\!> \rightarrow <\!N\!><\!C\!> \mid <\!C\!>$
5. $<\!C\!> \rightarrow \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9}$

Figura 33: Grammatica non ambigua per le espressioni aritmetiche dicono che un'espressione è un termine, seguito da zero o più coppie costituite da un `+` o un `-` e da un termine. Analogamente, la linea (2) dice che un termine è costituito da un termine seguito da `*` o `/` e da un fattore, oppure è un singolo fattore: detto altrimenti, un termine è un fattore seguito da zero o più coppie costituite da un `*` o un `/` e da un fattore. La linea (3) dice che i fattori sono numeri o espressioni racchiuse tra parentesi. Le linee (4) e (5) definiscono i numeri e le cifre come fatto in precedenza.

Il fatto che, alle linee (1) e (2), si siano utilizzate produzioni come

$$<E> \rightarrow <E> + <T>$$

anziché la produzione, apparentemente equivalente, $<E> \rightarrow <T> + <E>$, impone che i termini vengano raggruppati da sinistra: vedremo quindi che l'espressione `1-2+3` viene correttamente interpretata come `(1-2)+3`. Allo stesso modo, il termine `1/2*3` viene correttamente interpretato come `(1/2)*3`, e non erroneamente come `1/(2*3)`. La Figura 34 riporta l'unico albero di derivazione possibile per l'espressione `1-2+3`, rispetto alla grammatica di Figura 33. Si noti che `1-2` deve essere interpretata per prima come espressione: se avessimo raggruppato prima `2+3`, come in Figura 31(b), non avremmo trovato alcun modo, rispetto alla grammatica di Figura 33, per attaccare `1-` a questa espressione.

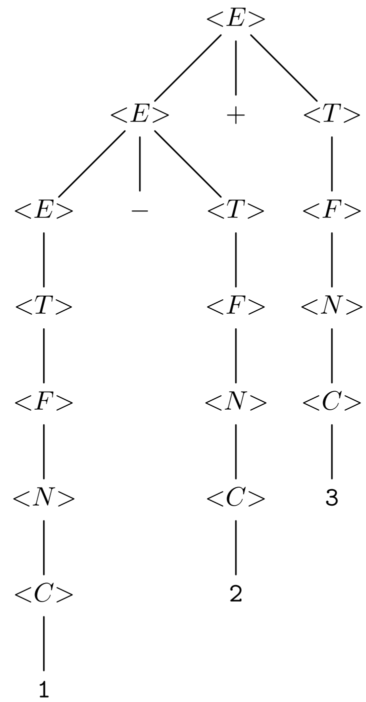


Figura 34: Albero di derivazione per l'espressione $\texttt{1} - \texttt{2} + \texttt{3}$ nella grammatica non ambigua di Figura 33

La distinzione tra espressioni, termini e fattori impone di raggruppare correttamente gli operatori di livelli di precedenza diversi: per esempio, l'espressione `1+2*3` ha un unico albero di derivazione, mostrato in Figura 35, in cui viene raggruppata per prima la sottoespressione `2*3`, come nell'albero di Figura 32(b) e non come nell'albero, scorretto, di Figura 32(a), in cui `1+2` viene raggruppata per prima.

Come nel caso delle parentesi bilanciate, non dimostriamo che la grammatica di Figura 33 è non ambigua.

Esercizi

3.4.1

In riferimento alla grammatica di Figura 33, mostrare l'unico albero di derivazione per ciascuna delle seguenti espressioni:

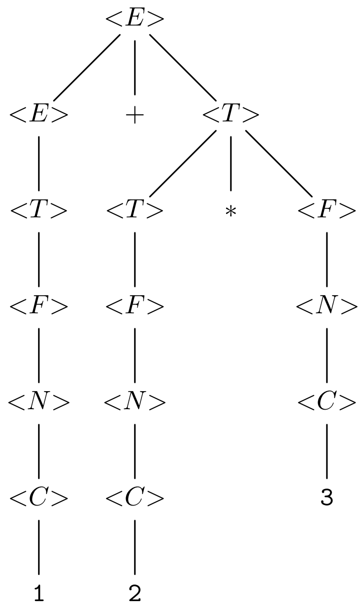


Figura 35: Albero di derivazione per `1` $+$ `2` $*$ `3` nella grammatica non ambigua di Figura 33

a) `(1+2)/3`

b) `1*2-3`

c) `(1+2)*(3+4)`.

3.4.2

Estendere la grammatica non ambigua delle espressioni per consentire l'uso degli operatori di confronto, `=`, `<=` e così via, tutti allo stesso livello di precedenza, un livello inferiore a quello di $+$ e $-$. Notare che gli operatori di confronto sono *non associativi*: non è cioè possibile raggruppare tre o più operatori di confronto. Per esempio, in Pascal `1<2<3` non è un'espressione legale.

3.4.3

Estendere la grammatica delle espressioni di Figura 33 in modo da includere il segno meno unario. Notare che tale operatore ha precedenza maggiore degli altri operatori: per esempio, `-2*-3` viene raggruppata come `(-2)*(-3)`.

3.4.4

Estendere la grammatica dell'Esercizio 3.4.2 in modo da includere gli operatori logici `AND`, `OR` e `NOT`. Dare a `AND` la stessa precedenza di `*`, a `OR` la stessa precedenza di `+` e a `NOT` una precedenza maggiore del $-$ unario. `AND` e `OR` sono operatori binari che raggruppano da sinistra.

3.4.5

In riferimento alla grammatica di Figura 15 del Paragrafo 3.1, non tutte le espressioni hanno più di un albero di derivazione. Dare alcuni esempi di espressioni che, in accordo a tale grammatica, hanno un solo albero di derivazione. Siete in grado di fornire una regola che indichi quando un'espressione ha un solo albero di derivazione?

3.4.6

La seguente grammatica definisce l'insieme delle stringhe (diverse da ϵ) costituite soltanto da sequenze di `0` e `1`.

$$<Stringa> \to <Stringa><Stringa> \mid \mathbf{0} \mid \mathbf{1}.$$

Rispetto a questa grammatica, quanti alberi di derivazione possiede la stringa `010`?

3.4.7

Determinare una grammatica non ambigua che definisca lo stesso linguaggio della grammatica dell'Esercizio 3.4.6.

3.4.8

Quanti alberi di derivazione ci sono per la stringa vuota rispetto alla grammatica (1)? Mostrare tre alberi di derivazione diversi per la stringa vuota.

3.5 Grammatiche ed automi

Sia le grammatiche che gli automi sono notazioni per esprimere linguaggi. È possibile che le grammatiche e gli automi siano notazioni equivalenti fra loro, ovvero che esprimano gli stessi linguaggi?

La risposta è "no": le grammatiche sono più potenti degli automi. Mostreremo il potere espressivo delle grammatiche in due passi. Per prima cosa mostreremo come un linguaggio esprimibile mediante un automa è anche descrivibile mediante una grammatica. Successivamente, mostreremo un linguaggio che può essere descritto da una grammatica, ma non da un automa.

In realtà, in queste note, ci stiamo limitando a considerare solo alcuni aspetti della teoria generale dei linguaggi formali. Il quadro generale è riassunto dalla seguente tabella

LINGUAGGI	**AUTOMI**	**GRAMMATICHE**	
Regolari	Automi a stati finiti	Regolari:	$A \to a \mid aB$ con $a \in \Lambda, A, \in V$
Liberi da contesto	Automi a pila	Libere:	$A \to \alpha$ con $A \in V, \alpha \in (\Lambda \cup V)^+$
Generali	Automi di Turing	Generali:	$\beta \to \alpha$ con $\alpha, \beta \in (\Lambda \cup V)^+$

In verticale la relazione valida è di inclusione propria. Ovvero i linguaggi regolari sono un sottoinsieme proprio dei linguaggi liberi da contesto e i linguaggi liberi da contesto sono un sottoinsieme proprio dei linguaggi generali.

Per verificare l'inclusione è sufficiente osservare la struttura delle produzioni delle corrispondenti grammatiche: è evidente che la struttura delle produzioni delle grammatiche regolari è una restrizione della struttura delle grammatiche libere e così via. Per dimostrare invece che l'inclusione è stretta è necessario trovare un controesempio.

Su ogni riga della tabella possiamo invece trovare la classe di linguaggi, il tipo di automa che riconosce tale classe e il tipo di grammatiche equivalenti, ovvero che generano linguaggi della stessa classe.

Rispetto alla tabella dimostreremo parte di una equivalenza tra automi e grammatiche, ovvero che esiste sempre una grammatica regolare che genera il linguaggio riconosciuto da un automa a stati finiti, e una inclusione propria, ovvero che esiste un linguaggio libero da contesto che non è regolare. Per fare questo faremo vedere che per un linguaggio di cui siamo in grado di esibire una grammatica libero che lo genera non può esistere un automa a stati finiti che lo riconosce.

Simulazione degli automi mediante grammatiche

L'idea per dimostrare che le grammatiche sono almeno "potenti" quanto gli automi è quella di definire, dato un generico automa, una grammatica che ne *simula* il comportamento, ovvero una grammatica che permetta di costruire un albero di derivazione per tutte e sole le stringhe accettate dall'automa. Tale costruzione è spiegata di seguito.

Dato un generico automa non deterministico A, costruiamo una grammatica associata G_A come segue:

- i simboli terminali di G_A coincidono con quelli dell'alfabeto Λ di A;
- le categorie sintattiche di G_A coincidono con gli stati di A;
- il simbolo iniziale di G_A coincide con lo stato iniziale di A;
- le produzioni di G_A sono assegnate nel modo seguente: per ogni transizione in A dallo stato s allo stato r etichettata con il simbolo a esiste una produzione:

  $$< s > \rightarrow \ \mathbf{a} < r > .$$

  Inoltre, se r è uno stato di accettazione di A, esiste oltre la precedente anche la produzione:

  $$< s > \rightarrow \ \mathbf{a}.$$

  Infine, se lo stato iniziale s è di accettazione, esiste anche la produzione:

  $$< s > \rightarrow \ \epsilon.$$

Ad esempio, la grammatica che simula l'automa di Figura 13(b) è data dalla seguenti produzioni:

$< 0 > \rightarrow \ \mathbf{a} < 0 > \mid \ \mathbf{b} < 0 > \mid \ \mathbf{c} < 0 > \mid \ \mathbf{a} < 1 >$

$< 1 > \rightarrow \ \mathbf{b} < 2 >$

$< 2 > \rightarrow \ \mathbf{c} < 3 > \mid \ \mathbf{c}$

$< 3 > \rightarrow \ \mathbf{a} < 3 > \mid \ \mathbf{b} < 3 > \mid \ \mathbf{c} < 3 > \mid \ \mathbf{a} \ \mid \ \mathbf{b} \ \mid \ \mathbf{c}$

È piuttosto immediato convincersi che, per ogni stringa t, esiste un cammino sul grafo di un automa A etichettato con t se e solo se esiste un albero di derivazione per t nella grammatica G_A. La dimostrazione di questo fatto è omessa in queste note.

Si noti che, per simulare un automa non deterministico, è sufficiente usare solo produzioni del tipo $< A > \rightarrow \mathbf{a} < B >$, oppure del tipo $< A > \rightarrow \mathbf{a}$. Le grammatiche che usano solo produzioni dei due tipi menzionati sono dette *grammatiche regolari*. Si dimostra assai semplicemente che le grammatiche regolari e gli automi sono equivalenti, nel senso che permettono di definire la stessa classe di linguaggi. Quindi, sappiamo per il momento che:

a) se un linguaggio è riconosciuto da un automa, allora è anche generato da una grammatica libera;

b) se un linguaggio è riconosciuto da un automa, allora è anche generato da una grammatica regolare, e viceversa.

Passiamo ora a dimostrare che il viceversa di a) è falso.

Un linguaggio con una grammatica ma non con un automa

Mostreremo adesso che le grammatiche sono più potenti degli automi, definendo un particolare linguaggio che è generato da una grammatica ma non è riconosciuto da nessun automa. Il linguaggio, che chiamiamo E, è l'insieme delle stringhe consistenti di uno o più `0` seguiti da un ugual numero di `1`:

$$E = \{\texttt{01}, \texttt{0011}, \texttt{000111}, \ldots\}$$

Per descrivere le stringhe di E esiste una notazione con esponenti: l'espressione s^n, dove s è una stringa e n è un intero, significa $ss \cdots s$ (n volte), ovvero, s concatenata con se stessa n volte. Quindi

$$E = \{\texttt{0}^1\texttt{1}^1, \texttt{0}^2\texttt{1}^2, \texttt{0}^3\texttt{1}^3, \ldots\}$$

ovvero, usando una notazione compatta,

$$E = \{\texttt{0}^n\texttt{1}^n \ \mid \ n \geq 1\}$$

Per prima cosa descriviamo E con la grammatica seguente:

(1) $<S> \to \mathbf{0}<S>\mathbf{1}$

(2) $<S> \to \mathbf{01}$

L'uso della produzione base, (2), esprime il fatto che `01` appartiene a $L(<S>)$. Al secondo passo possiamo usare la produzione (1), con `01` al posto di $<S>$ nel corpo, ottenendo $\texttt{0}^2\texttt{1}^2$ per $L(<S>)$. Un'ulteriore applicazione di (1) con $\texttt{0}^2\texttt{1}^2$ al posto di $<S>$ mostra che $\texttt{0}^3\texttt{1}^3$ appartiene a $L(<S>)$ e così via. In generale, $\texttt{0}^n\texttt{1}^n$ richiede l'uso della produzione (2), seguito da $n-1$ usi della produzione (1). Dato che non ci sono altre stringhe che possiamo generare con queste produzioni, possiamo affermare che $E = L(<S>)$.

Dimostrazione che E non può essere definito mediante automi

Vogliamo far vedere che E non può essere descritto mediante un automa a stati finiti deterministico.

Supponiamo, per assurdo, che E sia il linguaggio riconsociuto dall'automa a stati finiti A e che A abbia m stati. Consideriamo che cosa accade quando A riceve in ingresso $\texttt{000}\cdots$. Chiamiamo lo stato iniziale dell'automa A con il nome s_0. Sull'ingresso `0`, A deve avere una transizione uscente da s_0 verso un altro stato, che chiamiamo s_1. Da questo stato un altro `0` porta A in uno stato che chiamiamo s_2 e così via. In generale, dopo aver letto `0` per i volte, A si trova nello stato s_i, come mostrato dalla Figura 36.

Figura 36: L'automa A con alcuni 0 in ingresso

Avevamo supposto che A avesse esattamente m stati mentre ci sono $m+1$ stati in $s_0, s_1, \ldots, s_m$. Quindi non è possibile che tutti questi stati siano diversi. Ci devono essere almeno due interi distinti i e j, nell'intervallo da 0 a m, tali che s_i e s_j sono in realtà lo stesso stato. Assumendo che i sia il più piccolo tra i e j, il cammino di Figura 36 deve avere almeno un ciclo, come mostrato dalla Figura 37. Ovviamente ci potrebbero essere molti più cicli e ripetizioni di stati di quelli suggeriti dalla Figura 37. Notiamo anche che i potrebbe essere uguale a 0, nel qual caso il cammino da s_0 a s_i mostrato in Figura 37 è formato da un unico nodo. Analogamente, s_j potrebbe essere s_m, in questo casi il cammino da s_j a s_m è formato da un unico nodo.

La Figura 37 implica che l'automa A non può "ricordare" quanti `0` ha visto. Se è nello stato s_m, potrebbe aver visto esattamente m caratteri `0`, quindi, se partiamo dallo stato s_m e vediamo esattamente m caratteri `1` in ingresso, dobbiamo finire in uno stato di accettazione, come suggerito dalla Figura 38.

Supponiamo di dare in ingresso all'automa A una stringa di $m - j + i$ caratteri `0`. Guardando la Figura 37 vediamo che i caratteri `0` portano A da s_0 a s_i, che è lo stesso stato di s_j. Vediamo anche

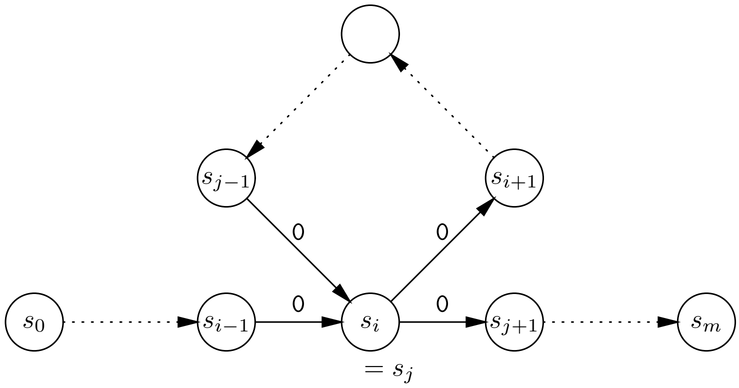


Figura 37: Il cammino di Figura 36 deve avere un ciclo

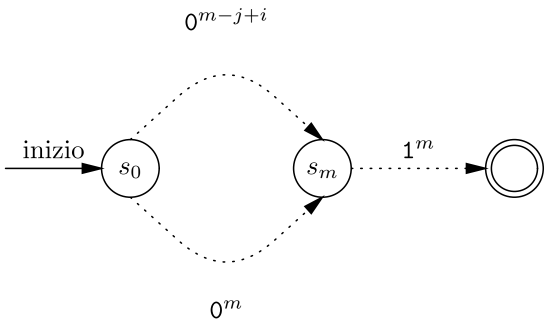


Figura 38: L'automa A non puo sapere se ha visto m caratteri 0 oppure $m - j + i$ caratteri 0

che $m - j$ caratteri 0 portano A da s_j a s_m. Quindi, $m - j + i$ caratteri 0 portano A da s_0 a s_m, come mostrato dal cammino in alto nella Figura 38.

Quindi $m - j + i$ caratteri 0 seguiti da m caratteri 1 portano A da s_0 ad uno stato di accettazione. Detto in altro modo, la stringa $0^{m-j+i}1^m$ appartiene al linguaggio di A. Poiché j è maggiore di i, questa stringa contiene un numero maggiore di 1 che di 0 e quindi non appartiene al linguaggio E. Possiamo concludere che il linguaggio di A non è uguale a E, come volevamo dimostrare.

Poiché siamo partiti dall'assunzione che il linguaggio E corrisponde a un automa finito deterministico e siamo arrivati a derivare una contraddizione, possiamo concludere che l'assunzione è falsa; cioè che non esiste nessun automa finito deterministico per E.

Il linguaggio $\{0^n1^n \mid n \geq 1\}$ è uno tra gli infiniti linguaggi che possono essere specificati da una grammatica, ma non da un automa.

3.6 Pumping Lemma

La dimostrazione che al linguaggio E non corrisponde un automa finito deterministico usa una tecnica conosciuta col nome di *principio delle buche dei piccioni* (*pigeonhole principle*), definito nel modo seguente.

"Se $m + 1$ piccioni si sistemano in m buche, allora ci deve essere almeno una buca con due piccioni".

Nel nostro caso, le buche sono gli stati dell'automa A e i piccioni sono gli $m + 1$ stati in cui si trova A dopo aver visto zero, uno, due e così via fino a m caratteri 0 in ingresso.

Si noti che m deve essere finito per poter applicare il principio delle buche dei piccioni.

Per formalizzare completamente quanto appena discusso è necessario far ricorso a un famoso teorema dei linguaggi formali: il *pumping lemma* dimostrato da Scott e Robin nel 1959. Nel seguito, data una stringa $w = a_1 \dots a_k$, indicheremo con $|w|$ la sua lunghezza, ovvero k (si noti che $|\epsilon| = 0$).

Pumping Lemma
Per ogni linguaggio L che sia regolare, esiste una costante n tale che ogni sequenza w di L, di lunghezza maggiore o uguale a n è esprimibile come

$$w = xyz$$

tali che:

- $y \neq \epsilon$
- $|xy| \leq n$
- $\forall i \geq 0.\ xy^i z \in L$

Dimostrazione
Osserviamo innanzitutto che se L è regolare e *finito*, il Lemma vale banalmente. Sia infatti

$$m = max\{\ell \mid \ell = |w| \wedge w \in L\}$$

Basta allora considerare $n = m + 1$ e l'asserto vale poiché non esiste alcuna stringa w in L con $|w| \geq n$. Sia dunque L un linguaggio regolare e infinito. Allora esiste un automa a stati finiti che lo riconosce. Supponiamo che tale automa abbia n stati. Poiché L è infinito, possiamo scegliere una stringa

$$w = a_1 a_2 \dots a_m$$

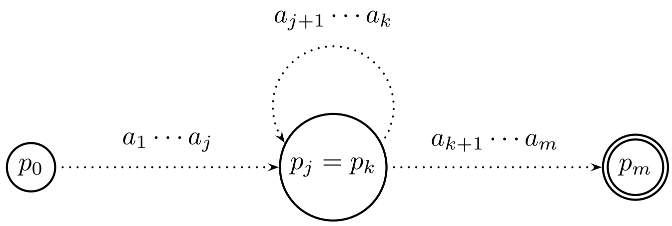
con $m > n$. Sia il cammino di accettazione di w

$$p_0 p_1 \cdots p_m$$

Poiché il numero di stati nel cammino è strettamente maggiore del numero di stati dell'automa (essendo $m > n$), debbono esistere j e k, con $0 \leq j < k \leq n$ tali che p_j e p_k sono lo stesso stato. In altre parole il cammino può essere riscritto come segue

$$p_0 \cdots p_j \cdots p_k \cdots p_n \cdots p_m$$

o rappresentato graficamente come



Poiché $j < k$ la stringa $a_{j+1} \dots a_k$ non è vuota e poiché $k \leq n$ la lunghezza della stringa $a_1 \dots a_k$ è al più n. È evidente inoltre che il ciclo che parte da p_j e ritorna a p_j può essere percorso zero o più volte e, di conseguenza, anche tutte le stringhe del tipo $xy^i z$ sono riconosciute dall'automa, per ogni i (si noti che la stringa w si ottiene per $i = 1$). Abbiamo allora $w = xyz$ con:

- $x = a_1 \dots a_j$
- $y = a_{j+1} \dots a_k$
- $z = a_{k+1} \dots a_m$

Le considerazioni precedenti ci consentono di affermare che:

- $y \neq \epsilon$
- $|xy| \leq n$
- $\forall i \geq 0.\ xy^i z \in L$ $\square$

Il lemma può essere utilizzato per dimostrare che un linguaggio *non* è regolare utilizzandolo nella sua versione contropositiva. Ricordando infatti che $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$ possiamo enunciare la contropositiva del lemma come:

Se $\forall n.\ \exists w \in L.$

$|w| \geq n \quad \wedge$

$\forall x, y, z.\ y \neq \epsilon \wedge |xy| \leq n \wedge \exists i \geq 0.\ xy^i z \notin L$

allora L non è regolare oppure L non è infinito.

Ma se dimostriamo la premessa e sappiamo che L è infinito possiamo concludere che L non è regolare.

Riprendendo l'esempio precedente

$$E = \{0^m 1^m \mid m \geq 1\}$$

dato un qualsiasi n prendiamo

$$w = 0^n 1^n$$

e consideriamo

$$w = xyz$$

Poiché $|xy| \leq n$ e $y \neq \epsilon$ abbiamo $y = 0^k$, ma per $i = 2$:

$$xy^i z = a^{n-k} a^{2k} b^n \notin L$$

poiché $n - k + 2k = n + k > n$ (k è positivo perché $y = 0^k \neq \epsilon \Rightarrow k > 0$).

Consideriamo ora il linguaggio

$$L = \{a^k b^m \mid k > 0 \wedge m > k\}$$

Dato un qualsiasi n prendiamo

$$w = a^n b^{2n}$$

e consideriamo

$$w = xyz$$

Poiché $|xy| \leq n$ e $y \neq \epsilon$ abbiamo $y = a^k$, e quindi:

$$xy^i z = a^{n-k}(a^k)^i b^{2n} = a^{n-k+ki} b^{2n}$$

Ma per i sufficientemente grande abbiamo che $n - k + ki > 2n$ e quindi $xy^i z \notin L$.

3.7 Note Bibliografiche

Le grammatiche libere da contesto sono state studiate inizialmente da Chomsky [1956] come formalismo per descrivere i linguaggi naturali. Formalismi simili sono stati usati per definire la sintassi di due tra i primi linguaggi di programmazione importanti: il Fortran (Backus et al. [1957]) e l'Algol 60 (Naur [1963]). Come conseguenza, le grammatiche libere da contesto sono spesso citate come grammatiche in Backus-Naur Form (BNF). Per uno studio più approfondito di queste grammatiche e delle loro applicazioni si consulti Hopcroft e Ullman [1979], oppure Aho, Sethi e Ullman [1986].

Aho, A. V., R. Sethi, e J. D. Ullman. *Compiler Design: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusets, 1986.

Backus, J. W.. "The FORTRAN automatic coding system", *Proc. AFIPS Western Joint Computer Conference*, pp. 188–198, Spartan Books, Baltimore, 1957.

Chomsky, N.. "Three models for the description of language", *IRE Trans. Information Theory* **IT-2**:3, pp. 113–124, 1956.

Hopcroft, J. E., e J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, Massachusets, 1979.

Naur, P. (a cura di). “Revised report on the algorithmic language Algol 60”, *Communications of the ACM*, **6**:1, pp. 1–17, 1963.