Strumenti per la definizione della sintassi dei Linguaggi di Programmazione[1]

Luca Tesei

A. A. 2004/2005

[1]Queste note intendono coprire gli argomenti della prima parte del corso di Programmazione e sono parzialmente tratte dal libro "Compilers: Principles, Techniques, and Tools" di Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullman, Addison-Wesley Pub

Capitolo 1

Linguaggi formali

Cominciamo con alcune nozioni di base sui linguaggi formali. Nei capitoli successivi vedremo diversi formalismi per specificarli e tratteremo anche della loro potenza espressiva.

1.1 Stringhe e linguaggi

Introduciamo alcune convenzioni che serviranno da qui in poi. I termini *alfabeto* o *classe di caratteri* denotano un qualsiasi insieme *finito* di simboli. Ad esempio l'insieme $\{0, 1\}$ è l'alfabeto binario. ASCII e Unicode sono esempi di alfabeti molto utilizzati nei computer. In genere useremo Σ (o in alcuni testi ed esercizi Λ) per denotare un alfabeto.

Una *stringa su un alfabeto* è una sequenza *finita* di simboli presi dall'alfabeto. Nell'ambito della teoria dei linguaggi i termini "parola" o "frase" sono sinonimi di "stringa". Stringhe generiche verranno nel seguito indicate con le variabili $s, s', s'', \ldots, s_0, s_1, s_2, \ldots$ oppure $x, y, z, w, \ldots, x', x'', \ldots, x_0, x_1, \ldots$ La lunghezza di una stringa s verrà indicata con $|s|$ ed è il numero di simboli che costituiscono la sequenza s. Ad esempio, `banana` è una stringa lunga 6. La *stringa vuota* è denotata con ϵ (o, in alcuni testi ed esercizi λ) e ha lunghezza 0. I seguenti sono altri termini usati in questo contesto con le relative definizioni:

- *Prefisso di s* Una stringa ottenuta togliendo zero o più simboli dalla fine di s. Es. `ban` è prefisso di `banana`.
- *Suffisso di s* Una stringa ottenuta togliendo zero o più simboli dalla testa di e. Es. `nana` è suffisso di `banana`.
- *Sottostringa di s* Una stringa ottenuta togliendo da s sia un suo suffisso che un suo prefisso. Es. `ana` è una sottostringa di `banana`. Si noti che ogni prefisso o suffisso di s è anche una sottostringa di s (ma non è vero il contrario). Si noti anche che s stessa è anche suo prefisso, suffisso e sottostringa.

- *Prefisso, suffisso e sottostringa propri di s* Una qualsiasi stringa non vuota x che sia, rispettivamente, un prefisso, un suffisso o una sottostringa di s e tale che $s \neq x$.

- *Sottosequenza di s* Una qualsiasi stringa ottenuta cancellando zero o più caratteri, non necessariamente contigui, da s. Es. `baaa` è una sottosequenza di `banana`.

Il termine *linguaggio* denota un qualsiasi insieme (finito o infinito) di stringhe su un certo alfabeto fissato Σ. Questa definizione è molto generale: ad esempio l'insieme vuoto ($\emptyset$ o $\{\}$) è un linguaggio (chiamato *linguaggio vuoto*) secondo questa definizione. Un altro linguaggio particolare è il linguaggio $\{\epsilon\}$ (o $\{\lambda\}$), che contiene una sola stringa, la stringa vuota. Altri esempi possono essere: tutti i programmi Pascal sintatticamente ben formati o addirittura tutte le frasi in italiano che sono grammaticalmente corrette. Tuttavia c'è subito da notare che quest'ultimo linguaggio è molto difficile da definire precisamente, mentre altri linguaggi non banali possono essere definiti matematicamente (come faremo noi con i linguaggi di programmazione tramite le grammatiche). Come ultima cosa si noti che la definizione che abbiamo dato non richiede che sia associato nessun significato particolare alle stringhe del linguaggio. La disciplina che si occupa dei metodi per dare significato alle stringhe di un linguaggio viene chiamata generalmente *semantica*. Vedremo uno di questi metodi nella seconda parte del corso.

Definiamo alcune operazioni di base sulle stringhe. Se x e y sono due stringhe, allora la *concatenazione* di x ed y, scritta come xy, è una stringa ottenuta attaccando y in fondo a x. Es. $x =$ `buon` e $y =$ `giorno` $xy =$ `buongiorno`. L'elemento neutro per la concatenazione è la stringa vuota: $\epsilon s = s\epsilon = s$ per ogni stringa s. Se pensiamo alla concatenazione come ad un prodotto, possiamo definire l'analogo dell'elevamento a potenza come segue:

- $s^0 = \epsilon$ per ogni stringa s

- $s^i = ss^{i-1}$ per ogni stringa s e per ogni $i > 0$

Ad esempio $\texttt{otto}^0 = \epsilon$, $\texttt{otto}^1 =$ `otto`, $\texttt{otto}^3 =$ `ottoottootto`. Si noti, inoltre, che anche una singola lettera è una stringa e quindi l'elevamento a potenza può essere usato anche in espressioni del tipo a^n. Specificando il valore di n otteniamo una stringa di tutte a lunga n.

1.2 Operazioni sui linguaggi

Ci sono diverse importanti operazioni che possono essere applicate ai linguaggi. Per quanto riguarda questo corso ci limiteremo a considerare le seguenti:

- *Unione* $L_1 \cup L_2 = \{s \mid s \in L_1 \vee s \in L_2\}$

- *Concatenazione* $L_1L_2 = \{s_1s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$
- *Esponenziazione* $L^0 = \{\epsilon\}$, $L^i = LL^{i-1}$ se $i > 0$
- *Chiusura (o stella) di Kleene* $L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \cdots$
- *Chiusura (o stella) positiva di Kleene* $L^+ = \bigcup_{i=1}^{\infty} L^i$

Esempio 1.1 *Sia L l'insieme $\{A, B, \ldots, Z, a, b, \ldots, z\}$ e D l'insieme $\{0, 1, \ldots, 9\}$. L e D possono essere visti in due modi: come alfabeti finiti o come linguaggi (finiti) composti da parole che sono tutte lunghe un carattere. Vediamo alcuni linguaggi che possono essere definiti a partire da L e D tramite le operazioni che abbiamo appena introdotto:*

- *$L \cup D$ è l'insieme delle lettere e delle cifre*
- *LD è l'insieme di tutte le stringhe di lunghezza 2 formate da una lettera seguita da una cifra*
- *L^4 contiene tutte le stringhe di lunghezza 4 che sono formate da lettere*
- *L^* è un insieme infinito di stringhe ognuna delle quali ha una lunghezza finita ed è composta da lettere. Questo linguaggio contiene, per definizione, anche la stringa vuota ϵ.*
- *$L(L \cup D)^*$ è l'insieme di tutte le stringhe di lunghezza strettamente maggiore di 0, che iniziano con una lettera e sono composte, dopo la prima lettera, da cifre o lettere.*
- *D^+ è l'insieme di tutte le stringhe di cifre formate da almeno una cifra*

Si noti che la stella di Kleene, in entrambe le sue versioni, può, e lo sarà spesso, essere applicata ad alfabeti. Quindi, secondo la definizione, dato in generale un certo alfabeto non vuoto Σ, Σ^* rappresenta tutte le stringhe di lunghezza finita, compresa quella vuota, che si possono formare usando i simboli di Σ. Σ^* è sempre un insieme infinito, ma che contiene parole di lunghezza finita. Inoltre $\Sigma^+ = \Sigma^* - \{\epsilon\}$.

Esempio 1.2 *Sia $\Sigma = \{0, 1\}$. Si ha che*

$$\Sigma^* = \{\epsilon, 0, 0, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}$$

$$\Sigma^+ = \{0, 0, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, \ldots\}$$

Tramite queste nuove nozioni possiamo dare una definizione precisa di linguaggio.

Definizione 1.3 (Linguaggio) *Dato un alfabeto non vuoto Σ, un linguaggio su Σ è un qualsiasi sottoinsieme di Σ^*: $L \subseteq \Sigma^*$.*

Abbiamo già visto i linguaggi banali $\emptyset$ e $\{\epsilon\}$. Un altro linguaggio banale su Σ è Σ^*.

1.3 Espressioni regolari

In questa sezione definiamo un formalismo che ci permetterà di definire un certo tipo di linguaggi. Ad esempio potremo definire il linguaggio che contiene tutte le stringhe che possono essere viste come identificatori di un linguaggio di programmazione.

L'insieme delle espressioni regolari su un certo alfabeto Σ è *definito induttivamente* dalle regole specificate nel seguito. Ogni espressione regolare su Σ definisce in maniera univoca un linguaggio su Σ. La definizione induttiva specifica (usando l'induzione sulla struttura delle espressioni) anche come viene formato il linguaggio definito da ogni espressione regolare:

Definizione 1.4 (Espressioni Regolari) *Sia Σ un alfabeto finito. L'insieme delle espressioni regolari su Σ è costituito da* tutte e sole *le espressioni generate induttivamente come segue. Associamo ad ogni espressione anche il linguaggio denotato.*

1. *ϵ è una espressione regolare che denota il linguaggio $\{\epsilon\}$*
2. *Se a è un simbolo di Σ allora a è un'espressione regolare che denota il linguaggio $\{a\}$*
3. *Siano r ed s due espressioni regolari che denotano i linguaggi $L(r)$ ed $L(s)$ rispettivamente. Allora:*
   - *(i) $(r)|(s)$ è una espressione regolare che denota il linguaggio $L(r) \cup L(s)$*
   - *(ii) $(r)(s)$ è una espressione regolare che denota il linguaggio $L(r)L(s)$*
   - *(iii) $(r)^*$ è una espressione regolare che denota il linguaggio $L(r)^*$*
   - *(iv) (r) è una espressione regolare che denota il linguaggio $L(r)$*

Un linguaggio denotato da una espressione regolare viene chiamato insieme (o linguaggio) regolare.

Esempio 1.5 *Prendiamo $\Sigma = \{a, b\}$ e proviamo a sviluppare la definizione induttiva per capire come è fatto l'insieme di tutte le espressioni regolari su $\{a, b\}$. Per adesso non consideriamo il linguaggio denotato.*

Procediamo costruttivamente, cioè costruiamo l'insieme partendo dall'insieme vuoto e usando le regole. All'inizio abbiamo l'insieme vuoto. Utilizzando i casi base 1. e 2. otteniamo l'insieme di espressioni regolari $\{\epsilon, a, b\}$. I casi induttivi (3.) non danno luogo a nessuna espressione perché richiedono di avere già a disposizione una o due espressioni regolari per formarne delle altre.

Partendo dall'insieme costruito applichiamo il passo induttivo per ampliare l'insieme. Per comodità chiamiamo ER_n l'insieme costruito al passo n (n è un numero naturale). Poniamo $ER_0 = \{\epsilon, a, b\}$.

In generale, per costruire ER_{n+1}, dobbiamo prendere le espressioni che sono già in ER_n ed inserirle in ER_{n+1}. Poi, usiamo le regole i-iv per aggiungerne di nuove prendendo come espressioni r ed s (le variabili usate nella definizione) tutte le combinazioni possibili di quelle già costruite nei passi precedenti (cioè quelle che troviamo in ER_n).

Calcoliamo ER_1. Utilizzando la regola (i) e assegnando alle variabili r ed s tutte le combinazioni di valori possibili otteniamo le espressioni $(\epsilon)|(\epsilon)$, $(\epsilon)|(a)$, $(\epsilon)|(b)$, $(a)|(\epsilon)$, $(a)|(a)$, $(a)|(b)$, $(b)|(\epsilon)$, $(b)|(a)$ e $(b)|(b)$.

Da (ii), alla stessa maniera, si ottengono le espressioni $(\epsilon)(\epsilon)$, $(\epsilon)(a)$, $(\epsilon)(b)$, $(a)(\epsilon)$, $(a)(a)$, $(a)(b)$, $(b)(\epsilon)$, $(b)(a)$ e $(b)(b)$.

Da (iii) si ottengono le espressioni $(\epsilon)^*$, $(a)^*$ e $(b)^*$. Da (iv) si ottengono le espressioni (ϵ), (a) e (b). Quindi:

$$ER_1 = \left\{ \begin{array}{l} \epsilon, a, b, (\epsilon)|(\epsilon), (\epsilon)|(a), (\epsilon)|(b), (a)|(\epsilon), (a)|(a), (a)|(b), (b)|(\epsilon), \\ (b)|(a), (b)|(b), (\epsilon)(\epsilon), (\epsilon)(a), (\epsilon)(b), (a)(\epsilon), (a)(a), (a)(b), (b)(\epsilon), \\ (b)(a), (b)(b), (\epsilon)^*, (a)^*, (b)^*, (\epsilon), (a), (b) \end{array} \right\}$$

Si noti che ϵ, a e b all'inizio dell'elenco sono le espressioni ereditate da ER_0.

Per calcolare ER_2 basta ripetere i passi i-iv prendendo però, questa volta, tutte le espressioni che abbiamo in ER_1 come possibili istanziazioni di r ed s. È facile rendersi conto che il processo può andare avanti all'infinito e può generare sempre nuove espressioni regolari su $\{a, b\}$. Quindi esistono infinite espressioni regolari su $\{a, b\}$ e sono tutte e sole quelle che possono venire generate, prima o poi, in questo processo.

La scrittura delle espressioni regolari secondo la definizione ha il difetto di generare espressioni con troppe parentesi, molte volte inutili. Per eliminare questo inconveniente evitiamo di mettere tra parentesi singoli simboli e assegniamo una precedenza ed una regola di associatività agli operatori che costruiscono espressioni regolari. In questo modo possiamo scrivere espressioni con meno parentesi, ma che hanno comunque un significato univoco. La precedenza è la seguente:

1. L'operatore unario * lega maggiormente. Quindi ad esempio ab^* sta per $(a)((b)^*)$, cioè la stella si riferisce solo a b e la a deve essere concatenata a b^*.

2. La concatenazione è il secondo operatore che lega maggiormente ed è associativa a sinistra (ad es. scrivendo abc si intende $((a)(b))(c)$)

3. L'operatore che lega meno di tutti è l'or ($|$) ed è anch'esso associativo a sinistra. Ad esempio, quindi, $a|ab$ va inteso come $a|(ab)$ (In quest'ultimo caso non sono state messe le parentesi intorno ad ogni simbolo).

Con queste convenzioni, per fare un altro esempio, l'espressione $(a)|((b)^*(c))$ può essere scritta come $a|b^*c$. Entrambe denotano il linguaggio delle stringhe che o sono a oppure sono una sequenza, eventualmente vuota, di b seguite da una c.

Esempio 1.6 *Vediamo ora un esempio di associazione del linguaggio denotato ad alcune espressioni regolari. Per trovarlo basta capire la struttura delle espressioni e applicare le regole della definizione. Sia* $\Sigma = \{a, b\}$.

- $a|b$ *denota il linguaggio* $\{a, b\}$
- $(a|b)(a|b)$ *denota il linguaggio* $\{aa, ab, ba, bb\}$, *denotabile anche con* $aa|ab|ba|bb$
- a^* *denota il linguaggio* $\{\epsilon, a, aa, aaa, aaaa, \ldots\}$
- $(a|b)^*$ *denota il linguaggio di tutte le stringhe formate dalla concatenazione di un numero qualsiasi di* a *e* b *messe in un qualsiasi ordine, più la stringa vuota. Ad esempio* $\epsilon, aab, baababa, a, b, bbbb, aaaa, abab, \ldots$.

Se due espressioni regolari denotano lo stesso linguaggio diciamo che sono *equivalenti*.

1.4 Specifica dei linguaggi tramite espressioni su insiemi

In questa sezione vediamo alcuni modi per specificare matematicamente alcuni linguaggi formali. Questo tipo di notazione sarà presente in tutto il corso. Abbiamo già visto che le espressioni regolari sono un modo semplice per specificare formalmente alcuni linguaggi, tuttavia esse hanno un potere espressivo limitato e molti linguaggi (i linguaggi non regolari) non possono essere specificati in questo formalismo.

Il mezzo più generale per dare una specifica precisa di molti linguaggi è rappresentato dalle *espressioni su insiemi*. Questo tipo di notazione è naturale poiché, effettivamente, i linguaggi formali non sono altro che *insiemi* di parole.

Un'espressione su insiemi non è altro che la classica notazione usata per rappresentare analiticamente gli elementi di un insieme:

$$\{x \in U \mid P(x)\}$$

dove U è un universo di oggetti dato a priori e P un predicato, una proprietà, che caratterizza tutti e soli gli elementi che fanno parte dell'insieme (cioè quegli x per cui $P(x)$ è vero).

Nel contesto dei linguaggi formali l'universo U è sempre Σ^*, dove Σ è l'alfabeto dal quale si prendono i simboli per formare le parole del linguaggio

che si sta definendo. Poiché Σ è spesso specificato preliminarmente e chiaro dal contesto, spesso l'universo viene sottinteso e la scrittura diventa $\{x \mid P(x)\}$.

La proprietà P può venire specificata tutta dopo il simbolo $|$, anche se spesso, specialmente nel campo della definizione di linguaggi, la struttura delle parole viene specificata parzialmente anche prima del simbolo $|$.

Per specificare la struttura possiamo usare i simboli di Σ, le operazioni su stringhe, le espressioni regolari e diverse variabili. Vediamo diversi esempi che illustrano i diversi modi di usare queste notazioni.

$$L = \{a^n\, b\, c \mid n > 0\}$$

In questo caso inferiamo che $\Sigma = \{a, b, c\}$ e che le parole del linguaggio sono tutte quelle che iniziano con una sequenza di a lunga almeno 1 ($n > 0$) e che poi continuano esattamente con una b seguita da una c. Esempi di parole del linguaggio: $abc, aabc, aaabc, aaaabc, \ldots$. Esempi di parole che non appartengono al linguaggio: $\epsilon, bc, aabac, aaaaa, aab, \ldots$

$$L = \{a^n\, b^*\, c^n \mid n \geq 0\}$$

In questo caso abbiamo usato anche un'espressione regolare. Il significato dei due esponenti n è quello usuale, cioè che in *tutte* le parole il numero di a e di c deve essere lo stesso e maggiore o uguale a zero ($n \geq 0$). All'interno delle a e delle c c'è una sequenza di b lunga anche zero (b^*). Esempi di parole nel linguaggio: ϵ (quando $n = 0$ e $b^* = \epsilon$), $ac, aacc, abc, aabcc, aabbbbbcc, b, aaabccc, \ldots$ Esempi di parole che non appartengono al linguaggio: $ab, aaabcc, abac, bbbccc, abccc, \ldots$

$$L = \{a^n\, (bb)^m\, c^k \mid n, m \geq 0, k > 0\}$$

Questo linguaggio potrebbe essere scritto anche come $a^*(bb)^*c^+$ o come $\{a^*\, b^{2m}\, c^k \mid m \geq 0, k > 0\}$. Le variabili n, m e k non sono legate l'una all'altra da nessun vincolo. Esempi di parole: $aaabbc, bbc, c, ccc, aaaaabbbbccc, \ldots$

$$L = \{a^n\, b^m\, c^k \mid n, m > 0, k = m + n\}$$

In questo caso le variabili n, m, k sono interdipendenti. Il vincolo $k = m + n$ indica che la somma fra il numero di a e il numero di b deve essere uguale al numero di c in ogni parola del linguaggio. Esempi: $aabbbcccc, abcc, abbbbbccccccc, \ldots$ Esempi di parole non del linguaggio: $c, abc, aabb, cc, \ldots$

$$L_1 = \{a^n\, b\, c^+ \mid n > 0\}$$
$$L = \{d\, x\, a^n \mid x \in L_1, n \geq 0\}$$

Qui abbiamo usato un sottolinguaggio L_1 per definire un altro linguaggio principale L. Il linguaggio L ha parole che cominciano con una d seguita da

una qualsiasi parola del linguaggio L_1 seguita a sua volta da una sequenza di a anche vuota. Si noti che la variabile n in L_1 e quella in L non hanno nessun rapporto tra di loro, poiché hanno *scope* diverso, cioè il loro significato è circoscritto all'interno delle parentesi graffe di ognuna delle due espressioni insiemistiche che le usano. Esempi: $daabccccaaaaaaaa, dabc, daabccca, \ldots$. Non appartenenti: $\epsilon, dbcaaa, daab, aaabcaaaa, \ldots$

$$L = \{s_1\, d\, s_2 \cdots s_k\, d \mid k \geq 0, s_i \in \{a,b,c\}^* \forall i \in \{1,\ldots,k\}\}$$

Abbiamo usato, in questo esempio, una sequenza numerata di stringhe (s_i) ognuna delle quali può essere una stringa qualunque di un altro linguaggio $\{a,b,c\}^*$ (potevamo usare anche un sottolinguaggio come nell'esempio precedente). Il fatto che k possa essere anche 0 indica che nel linguaggio è presente anche la stringa vuota. Se $k > 0$ allora alla fine di ogni s_i deve essere presente una d. Esempi: $\epsilon, aabbcd, d, dddd, aacdaabbdbcabdabacd, \ldots$ Non appartenenti: $abcac, bcdabac, \ldots$.

Infine, vedremo che spesso i linguaggi che specificheremo sono divisi per "casi", cioè il linguaggio che vogliamo descrivere risulta dall'*unione* di diversi linguaggi componenti ognuno dei quali rappresenta un caso. In questo caso è sufficiente utilizzare la normale unione insiemistica. Ad esempio:

$$L = \{a^n\, b\, c \mid n > 0\} \cup \{b^n\, c\, a \mid n \geq 0\} \cup \{d^+\, (c|b)\, a^m \mid m > 0\}$$

Si noti che la n del primo insieme e la n del secondo, ancora una volta, non hanno nessun rapporto: una qualsiasi potrebbe essere rinominata senza cambiare il linguaggio definito. Si noti inoltre che nell'ultimo insieme abbiamo usato l'or delle espressioni regolari. Esempi di stringhe: $abc, bca, ca, dc, ddb, \ldots$

Capitolo 2

Automi a stati finiti

In questo capitolo definiamo gli automi a stati finiti non deterministici e deterministici. Essi sono dei riconoscitori di stringhe di un certo linguaggio. Si ha che tutti i linguaggi regolari, cioè quelli denotabili da espressioni regolari, possono essere accettati anche da un automa a stati finiti (è vero anche il viceversa). Dopo la definizione degli automi vedremo un algoritmo importante: la costruzione dei sottoinsiemi per trasformare un automa non deterministico in deterministico.

2.1 Automi non deterministici

Definiamo la classe degli automi finiti non deterministici NFA (dall'inglese Non-deteriministic Finite Automata). Sia Σ un alfabeto finito.

Definizione 2.1 (NFA) *Un* NFA *su* Σ *è una tupla* $\langle S, \Sigma, move, s_0, F \rangle$ *dove:*

- S *è un insieme finito di* stati
- Σ *è l'alfabeto dei simboli*
- $s_0 \in S$ *è lo* stato iniziale
- $F \subseteq S$ *è l'insieme degli* stati di accettazione *o* stati finali
- $move: (S \times \Sigma) \longrightarrow \wp(S)$ *è una funzione che specifica per ogni stato* s *e per ogni simbolo* x *di* Σ *le* transizioni etichettate x *dallo stato* s *ad un insieme, anche vuoto, di stati di destinazione (c'è una transizione per ogni stato di destinazione).*

Un NFA può essere rappresentato graficamente tramite un diagramma in cui i cerchi sono gli stati, le frecce etichettate sono le transizioni, lo stato iniziale ha una freccia entrante con scritto *start* e gli stati finali hanno una

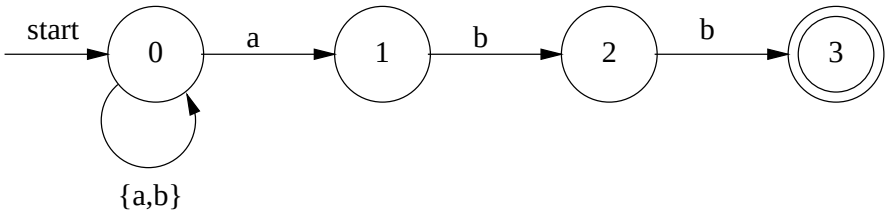


Figura 2.1: Un NFA.

doppia cerchiatura. Ogni freccia può essere etichettata da un simbolo di Σ o da un insieme di simboli di Σ. Le transizioni di quest'ultimo tipo sono un'abbreviazione che denota un insieme di transizioni, tutte con la stessa sorgente e la stessa destinazione, etichettate ognuna con un simbolo diverso dell'insieme.

Un esempio di NFA si trova in Figura 2.1. S è l'insieme $\{0,1,2,3\}$, $\Sigma = \{a,b\}$, lo stato iniziale è 0 e l'unico stato finale è 3 ($F = \{3\}$). Le frecce rappresentano le transizioni: ad esempio la freccia etichettata $\{a,b\}$ che parte dallo stato 0 e va nello stato 0 rappresenta due transizioni: una etichettata con a, l'altra con b ed entrambe che partono da 0 e ritornano in 0 (questo tipo di transizioni vengono chiamate anche *self-loop*). Esiste un'altra transizione uscente dallo stato 0 ed etichettata con a, ma è diversa dalla precedente poiché lo stato di destinazione è 1. Queste due transizioni indicano che $move(0,a) = \{0,1\}$. Inoltre $move(0,b) = \{0\}$, $move(1,a) = \{\}$, $move(1,b) = \{2\}$, $move(2,a) = \{\}$, $move(2,b) = \{3\}$ e $move(3,a) = move(3,b) = \{\}$.

Un NFA può riconoscere le stringhe di un certo linguaggio. Per chiarire in quale modo un NFA accetta una stringa definiamo la nozione di cammino etichettato.

Definizione 2.2 (Cammino etichettato) *Sia $N = \langle S, \Sigma, move, s_0, F\rangle$ un NFA. Un cammino etichettato di lunghezza $k \geq 0$ su N è una sequenza $s_0 \xrightarrow{x_0} s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} \cdots s_{k-1} \xrightarrow{x_{k-1}} s_k$ dove:*

- *s_0 è lo stato iniziale*
- *$\forall i \in \{0,1,\ldots,k-1\}$. $s_{i+1} \in move(s_i, x_i)$*

La stringa $x_0 x_1 \cdots x_{k-1}$ si chiama stringa associata al cammino *ed è in generale una stringa di Σ^*. Se $k = 0$ allora il cammino è costituito solo dallo stato iniziale e la stringa associata è la stringa vuota ϵ.*

Si noti che un cammino di lunghezza k consiste di una sequenza di $k+1$ stati dell'automa ed ha una stringa associata lunga k.

Definizione 2.3 (Stringa accettata) *Sia $N = \langle S, \Sigma, move, s_0, F\rangle$ un NFA. Una stringa $\alpha \in \Sigma^*$ è accettata dall'automa N* **se e solo se** *esiste un*

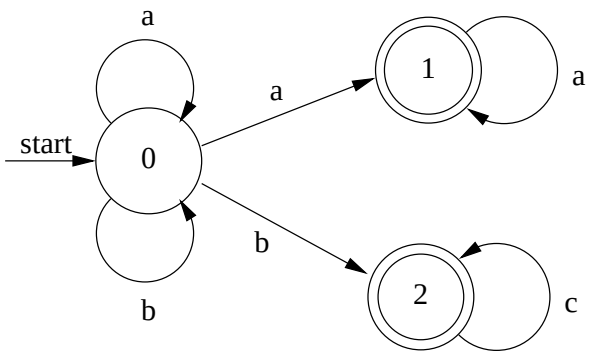


Figura 2.2: Un NFA.

cammino etichettato $s_0 \xrightarrow{x_0} s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} \cdots s_{k-1} \xrightarrow{x_{k-1}} s_k$ *su* N *tale che* α *è la stringa associata al cammino e lo stato* s_k *è uno stato finale (in formule* $s_k \in \mathcal{F} \wedge \alpha = x_0 x_1 \cdots x_{k-1}$*). Se lo stato iniziale è anche di accettazione allora anche la stringa vuota* ϵ *è accettata dall'automa.*

Possiamo ora introdurre la nozione di linguaggio accettato dall'automa.

Definizione 2.4 (Linguaggio accettato) *Sia* $N = \langle S, \Sigma, move, s_0, F \rangle$ *un NFA. Il linguaggio accettato dall'automa è l'insieme*

$$L(N) = \{\alpha \in \Sigma^* \mid \alpha \text{ è accettata da } N\}$$

Esempio 2.5 *Consideriamo l'automa in Figura 2.1.*

$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

è un cammino etichettato la cui stringa associata è aabb. Lo stato in cui il cammino termina, 3, è anche uno stato finale e quindi questa stringa è accettata dall'automa.

Essendo l'automa non deterministico, tuttavia, esiste un altro cammino etichettato con la stringa precedente:

$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

in questo caso lo stato 0 non è finale e quindi questo cammino non porta all'accettazione della stringa. Si noti che, comunque, la definizione richiede che ci sia almeno *un cammino etichettato che termina in uno stato finale per determinare se la stringa associata è accettata o no.*

Facendo altri esempi e considerando la struttura dell'automa è facile convincersi che il linguaggio accettato è quello denotato da

$$(a|b)^* abb = \{s\, abb \mid s \in \{a, b\}^*\}$$

La costruzione di un cammino etichettato per una data stringa può essere vista come un algoritmo di riconoscimento di stringhe. Ad esempio, supponiamo che vogliamo controllare se la stringa *aba* fa parte del linguaggio

accettato dall'automa di Figura 2.2. Partiamo dallo stato iniziale e manteniamo un insieme di stati "correnti". All'inizio questo insieme è $\{0\}$ dato che 0 è l'unico stato in cui mi posso trovare all'inizio mentre sto cercando di costruire un cammino etichettato per la stringa *aba*.

Consideriamo ora il primo simbolo della stringa: a. A questo punto guardiamo quali sono gli stati in cui possiamo andare seguendo una transizione etichettata a uscente da uno qualsiasi degli stati correnti. L'insieme degli stati che posso raggiungere con la prima a è $\{0,1\}$ poichè dallo stato 0 ci sono due transizioni uscenti etichettate con a ($move(0,a) = \{0,1\}$). Questo è il nuovo insieme di stati correnti.

Eliminiamo il primo simbolo dalla stringa in esame e consideriamo quello seguente: b. A partire da ogni stato in $\{0,1\}$ controlliamo in quali stati posso andare seguendo transizioni etichettate con b. Si ha che $move(0,b) = \{0,2\}$ e $move(1,b) = \{\}$. Il nuovo insieme di stati correnti sarà quindi $\{0,2\}$.

Eliminiamo il simbolo b e passiamo al successivo e ultimo: a. Dato che $move(0,a) = \{0,1\}$ e $move(2,a) = \{\}$, si ha che il nuovo insieme degli stati correnti è $\{0,1\}$.

A questo punto la stringa di input è stata letta tutta e non ci rimane che controllare se nell'insieme di stati correnti sia presente almeno uno stato finale. Questo è vero poiché 1 è finale e quindi possiamo concludere che la stringa *aba* è accettata dall'automa.

Un cammino etichettato che porta all'accettazione della stringa è il seguente:

$0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 1$

Consideriamo invece la stringa *ac*. A partire da uno degli stati in $\{0\}$ possiamo arrivare, con una a, nell'insieme di stati $\{0,1\}$. Eliminiamo la prima a e consideriamo la c seguente. A questo punto si ha che $move(0,c) = \{\}$ e $move(1,c) = \{\}$. Entrambi sono vuoti. In una situazione come questa l'automa si dice *bloccato* poiché non può andare avanti a costruire cammini possibili per tutto l'input. Ricordando la definizione di stringa accettata vediamo che in questo caso non esiste nessun cammino etichettato per la stringa e che quindi l'automa non accetta.

Procedendo con altre prove si può inferire che il linguaggio accettato dall'automa in Figura 2.2 è

$$(a|b)^*(a^+|bc^*) = \{s\,a^n \mid s \in \{a,b\}^*, n > 0\} \cup \{s\,b\,c^n \mid s \in \{a,b\}^*, n \geq 0\}$$

2.2 Automi finiti deterministici

La definizione di automa che abbiamo dato è quella più generale, cioè quella di automa non deterministico. Diamo ora una caratterizzazione degli automi deterministici DFA (Deterministic Finite Automata).

Definizione 2.6 (Automa deterministico) *Un automa finito deterministico (DFA) è una tupla* $\langle S, \Sigma, move, s_0, F \rangle$ *dove:*

- S *è un insieme finito di stati.*
- Σ *è un alfabeto finito di simboli.*
- $move: (S \times \Sigma) \longrightarrow \wp(S)$ *è una funzione di transizione tale che per ogni stato* $s \in S$ *e per ogni simbolo* $x \in \Sigma$ *l'insieme degli stati* $move(s, x)$ *o è vuoto oppure contiene un solo stato.*
- $s_0 \in S$ *è lo stato iniziale.*
- $F \subseteq S$ *è l'insieme degli stati finali.*

In altre parole un automa è deterministico se da ogni stato non escono mai due transizioni etichettate con lo stesso simbolo.

La nozione di cammino etichettato e di accettazione per un DFA è analoga a quella di un NFA. Dato che in un DFA, dato uno stato e un simbolo, è possibile al più una transizione etichettata con quel simbolo, si ha che per ogni stringa accettata da un DFA esiste un unico cammino etichettato con la stringa e che termina in uno stato finale.

L'algoritmo di riconoscimento di una data stringa è uguale a quello di un NFA, ma in questo caso ad ogni passo l'insieme degli stati correnti contiene uno ed un solo stato.

Esempio 2.7 *Consideriamo l'automa disegnato in Figura 2.3. Esso è un automa deterministico che accetta lo stesso linguaggio dell'automa di Figura 2.1, cioè* $(a|b)^*abb$. *L'unico cammino di accettazione per la stringa aabb è*

$0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Il cammino di accettazione per la stringa abaabb è:

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

2.3 Costruzione dei sottoinsiemi

Gli automi deterministici e quelli non deterministici hanno lo stesso potere espressivo. Questo significa che se un linguaggio può essere accettato da un NFA allora esiste anche un DFA che lo accetta, e viceversa.

E' chiaro che l'algoritmo di riconoscimento di una certa stringa di un DFA è più immediato poichè non bisogna tener traccia di un insieme di stati (il cammino di accettazione, se esiste, è unico). D'altra parte, da un punto di vista di progettazione, il non determinismo rende più facile scrivere un automa e/o determinare che tipo di linguaggio accetta, oltre a permettere di scrivere automi più concisi.

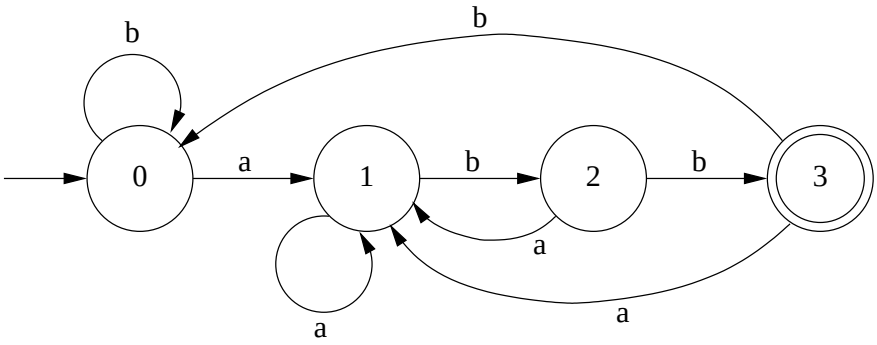


Figura 2.3: Un automa deterministico.

Basta ad esempio guardare i due automi delle Figure 2.1 e 2.3 che accettano lo stesso linguaggio. Il primo è non deterministico nello stato 0 e specifica in maniera naturale che si possono riconoscere a o b in qualsiasi numero ed ordine prima di passare ad una stringa finale obbligatoria abb. Il secondo invece deve esplicitare una sorta di backtracking negli stati: la prima a che incontra potrebbe essere quella della stringa finale obbligatoria e quindi l'automa entra nello stato 1. Se il simbolo seguente non è una b allora l'automa continua a ciclare nello stato 1. Nello stato 2, se il simbolo seguente non è l'ultima b l'automa ritorna nello stato 1 ad aspettare di leggere un'altra a possibile candidata ad essere il primo simbolo della stringa finale obbligatoria. Infine nello stato 3 l'automa deve ritornare nello stato 0 o 1 se ci sono ancora simboli b o a, rispettivamente. Questo perchè i simboli letti precedentemente potrebbero essere il prefisso di una stringa che poi terminerà con la stringa finale obbligatoria.

In questa sezione formalizziamo un algoritmo noto come *costruzione dei sottoinsiemi* (subset construction) che serve per costruire, a partire da un NFA dato, un DFA equivalente che simula il non determinismo, ma è deterministico.

Per specificare l'algoritmo useremo uno pseudo-codice in cui le strutture dati vere e proprie non saranno specificate (in ogni caso non è difficile implementare questi algoritmi in un linguaggio di programmazione: basta definire le opportune strutture dati e le varie funzioni/procedure che specifichiamo).

Algoritmo 2.1 (Subset construction) Costruzione di un DFA a partire da un NFA.

Input: Un NFA $N = \langle S, \Sigma, move, s_0, F \rangle$

Output: Un DFA $D = \langle DStates, \Sigma, Dtran, s'_0, F' \rangle$ equivalente a N e tale che $DStates \subseteq \wp(S)$.

Metodo: Costruiamo la funzione di transizione $Dtran$ simulando, attraverso

insiemi di stati di N, i comportamenti di N "in parallelo" dovuti al non determinismo.

L'algoritmo è uno dei classici algoritmi di calcolo di un punto fisso (il calcolo di un insieme definito induttivamente) e ha la proprietà di terminare sempre in un numero finito di passi. In pratica partiamo da un insieme di stati *DStates* contentente solo uno stato iniziale non marcato. Ad ogni passo selezioniamo uno stato non marcato da *DStates* e ne calcoliamo le transizioni uscenti aggiungendo a *DStates*, non marcati, eventuali nuovi stati trovati. Prima o poi, dato che il numero di stati possibili è finito (il numero di elementi di $\wp(S)$) non ci saranno più stati non marcati da considerare e l'algoritmo terminerà.

Operazioni utilizzate:

- $move: (\wp(S) \times \Sigma) \longrightarrow \wp(S)$ tale che $move(T, x) = \bigcup_{s \in T} move(s, x)$. In pratica si mettono in uno stesso insieme tutti gli stati raggiungibili con uno stesso simbolo x a partire da uno stato qualunque di T.

Algoritmo:

all'inizio $\{s_0\}$ è l'unico stato di *DStates* e non è marcato;
while c'è uno stato non marcato T in *DStates* **do begin**
  marca T;
  for each simbolo di input $x \in \Sigma$ **do begin**
    $U := move(T, x)$;
    if U non è in *DStates* **then**
      aggiungi U, non marcato, a *DStates*;
    $Dtran(T, x) := U$;
  end;
end;
lo stato iniziale di D è $\{s_0\}$;
gli stati finali di D sono tutti quelli che contengono almeno uno stato finale di N **end**

Esempio 2.8 *Consideriamo come automa N di partenza quello in Figura 2.1. Il linguaggio accettato, come sappiamo, è quello denotato dall'espressione regolare $(a|b)^*abb$. Applichiamo l'algoritmo della costruzione dei sottoinsiemi e troviamo il DFA D equivalente.*

Lo stato iniziale di D è per definizione $\{0\}$ poiché 0 è lo stato iniziale di N. Chiamiamo questo insieme, per convenienza, A. A è il primo stato non marcato che fa parte di DStates.

Alla prima iterazione selezioniamo per forza lo stato A e lo marchiamo. Calcoliamo:

- $move(A, a)) = \{0, 1\}$ *Chiamiamo questo nuovo stato* $B = \{0, 1\}$
- $move(A, b)) = \{0\} = A$

Con una b ritorniamo nello stato iniziale e con una a andiamo un uno stato nuovo B (non si trova attualmente in DStates) che, seguendo l'algoritmo, va inserito non marcato in DStates.

In generale, una funzione di transizione di un NFA o di un DFA può essere rappresentata anche con una tabella in cui le righe sono gli stati e le colonne sono i simboli dell'alfabeto. La cella ad una riga T e ad una colonna x della tabella è l'insieme di stati che risulta da $move(T, x)$.

La tabella che rappresenta la parte di Dtran calcolata fino a questo punto è la seguente:

Stato	*a*	*b*	*Marcato*
$A = \{0\}$	B	A	*Si*
$B = \{0, 1\}$			*No*

Procediamo scegliendo uno stato non marcato. Anche questa volta la scelta obbligata è B e calcoliamo:

- $move(B, a)) = \{0, 1\} = B$
- $move(B, b)) = \{0, 2\}$ *Chiamiamo questo nuovo stato* $C = \{0, 2\}$

Anche questa volta per il simbolo a non è stato generato nessuno stato nuovo, mentre per b è stato generato il nuovo stato C che inseriamo non marcato in DStates. La tabella parzialmente costruita fino a questo punto è la seguente:

Stato	*a*	*b*	*Marcato*
$A = \{0\}$	B	A	*Si*
$B = \{0, 1\}$	B	C	*Si*
$C = \{0, 2\}$			*No*

Continuando ad applicare l'algoritmo si arriva a generare un ulteriore stato $D = \{0, 3\}$ dopodiché non si generano più nuovi stati e l'algoritmo termina. La tabella finale è la seguente:

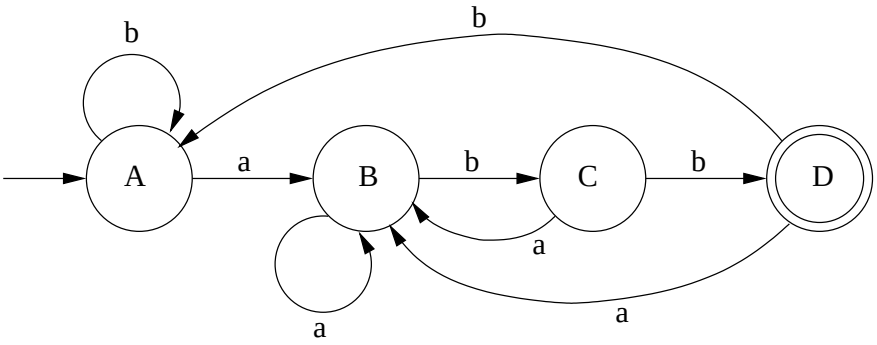


Figura 2.4: L'automa deterministico risultante dalla costruzione dei sottoinsiemi.

Stato	*a*	*b*	*Marcato*
$A = \{0\}$	B	A	*Si*
$B = \{0, 1\}$	B	C	*Si*
$C = \{0, 2\}$	B	D	*Si*
$D = \{0, 3\}$	D	A	*Si*

L'unico stato finale è D poiché è l'unico che contiene uno stato finale di N, cioè 3. L'automa è disegnato in Figura 2.4. Si noti che questo è lo stesso automa di Figura 2.3 a meno di ridenominazione degli stati (0=A, 1=B, 2=C, 3=D).

Per concludere osserviamo che in tutte le caselle della tabella dell'esempio abbiamo ottenuto un insieme di stati da inserire. Tuttavia questo non si verifica sempre: se otteniamo, per una certa casella, $move(T, x) = \{\}$ allora significa che nell'automa risultante D, a partire dallo stato T, non c'è nessuna transizione uscente etichettata con x.

2.4 Minimizzazione di un DFA

La facilità di simulazione di un DFA rispetto a quella di un NFA è compensata dal fatto che il DFA in genere ha un numero maggiore di stati. È lecito chiedersi, in quest'ottica, se è possibile, dato in certo DFA, trovarne uno equivalente, ma che abbia un numero minore di stati. La risposta è sì, ed esiste un algoritmo che trova un DFA equivalente ad un DFA dato e tale che l'automa risultato ha un numero *minimo* di stati. Qui minimo si riferisce agli stati necessari per poter accettare il linguaggio accettato dall'automa di partenza. Inoltre si ha che l'automa minimo è unico a meno di rinomi-

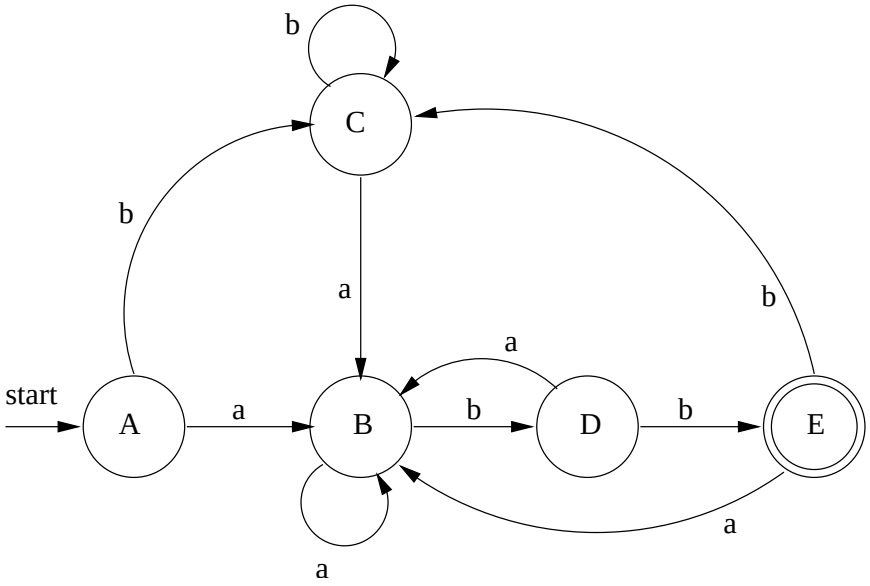


Figura 2.5: Un automa deterministico per $(a|b)^*abb$.

nare gli stati (in altre parole se trovo due automi minimi esiste sempre una funzione bigettiva fra gli stati dei due che rispetta tutte le transizioni).

In questo corso non vediamo questo algoritmo, ma è bene sapere che esiste la nozione di minimalità e di equivalenza di stati. Si osservi ad esempio l'automa deterministico di Figura 2.5. È facile convincersi che anche questo automa accetta il linguaggio $(a|b) * abb$ come quello risultante dalla costruzione dei sottoinsiemi in Figura 2.4. Il numero di stati di quest'ultimo, però, è minore del numero di stati dell'altro. L'algoritmo di minimizzazione, partendo dall'automa con più stati, avrebbe riconosciuto che gli stati 0 e 2 sono equivalenti e li avrebbe unificati in un unico stato dimostrando, inoltre, che l'automa di Figura 2.4 è minimo.

Capitolo 3

Grammatiche libere dal contesto

Ogni linguaggio di programmazione ha delle regole che prescrivono la struttura sintattica dei programmi ben formati del linguaggio. Ad esempio in Pascal un programma corretto è formato da blocchi, i blocchi sono formati da statements, gli statements da espressioni, le espressioni dai token e così via.

La sintassi dei linguaggi di programmazione può essere ed è convenientemente specificata tramite grammatiche libere dal contesto (*context-free grammars*). Lo stesso formalismo viene a volte chiamato anche notazione BNF (Backus-Naur Form).

Le grammatiche offrono vantaggi significativi sia ai progettisti dei linguaggi di programmazione che ai loro implementatori. Vediamo i principali:

- Una grammatica dà una specifica sintattica precisa e facile da capire di un linguaggio.
- Per alcune classi di grammatiche possiamo automaticamente costruire degli analizzatori efficienti che determinano se un certo programma sorgente è sintatticamente ben formato e, in caso positivo, ne rendono disponibile la struttura gerarchica ad albero. Inoltre, il processo di costruzione dell'analizzatore stesso riesce a rilevare ambiguità nella definizione della grammatica o ad individuare alcuni costrutti critici difficili da analizzare. Questo tipo di problemi possono facilmente passare inosservati nella fase iniziale di progetto di un linguaggio e questa possibilità è un valido supporto per i progettisti.
- Una grammatica ben progettata impone una struttura al linguaggio e questa struttura è utile per la traduzione, da parte del compilatore, dei costrutti in codice oggetto (sia intermedio che della macchina ospite). Questo processo di traduzione può essere fatto automaticamente dando le specifiche giuste ai generatori automatici di analizzatori.

- Un linguaggio può evolvere nel tempo acquisendo nuovi costrutti e/o eseguendo nuove operazioni. Questi nuovi costrutti possono essere aggiunti più facilmente se il linguaggio è stato implementato sulla base di una grammatica.

In questo capitolo introdurremo le grammatiche libere dal contesto e le relative nozioni di albero di derivazione, derivazioni, ambiguità.

3.1 Grammatiche libere dal contesto

Diamo ora una definizione precisa di grammatica libera dal contesto.

Definizione 3.1 *Una* Grammatica libera dal contesto *è una tupla* $G = \langle \Sigma, V, S, P \rangle$ *dove:*

- Σ *è un insieme finito dei simboli di alfabeto o* Simboli Terminali
- V *è un insieme finito di simboli che rappresentano* Categorie Sintattiche *o simboli* non terminali
- $S \in V$ *è un simbolo di categoria sintattica indicato come* iniziale *o* principale
- P *è un insieme finito di regole, chiamate* Produzioni*, della forma*

$$A \to X_1 X_2 \cdots X_k$$

dove:

  - $A \in V$ *è la* testa *o parte sinistra della produzione*
  - $\forall i \in \{1, \ldots, k\}.\ X_i \in (V \cup \Sigma)$. *La stringa* $X_1 X_2 \cdots X_k \in (V \cup \Sigma)^*$ *si chiama* corpo *o parte destra della produzione.*

Il corpo di una produzione può anche essere vuoto. In questo caso la produzione viene scritta $A \to \epsilon$.

In alcuni testi l'operatore `::=` *è usato a posto della freccia* $\to$ *nella scrittura delle produzioni. Le due notazioni sono equivalenti.*

Il compito principale di una grammatica è quello di generare stringhe di simboli terminali. Il processo di generazione delle stringhe può avvenire tramite costruzione di alberi di derivazione o, equivalentemente, derivazioni.

Prima di tutto fissiamo alcune convenzioni di notazione che ci permetteranno di illustrare più chiaramente i concetti che introdurremo via via nel seguito. Associamo degli insiemi di simboli a certi tipi di oggetti formali che tratteremo spesso. In questo modo eviteremo di dover specificare, ogni volta che useremo un simbolo nuovo, che cosa quel simbolo sta ad indicare.

1. I seguenti simboli indicano simboli **terminali** della grammatica:

   - Lettere minuscole all'inizio dell'alfabeto:

   $$a, b, c, \ldots, a', a'', \ldots, a_1, a_2, \ldots$$

   - Simboli di operatori: $+$, $-$, $*$, ...
   - Simboli di punteggiatura come virgole, punti e virgola, due punti, punti, ...
   - Le cifre 0,1,...,9.
   - Stringhe in grassetto come **id** o **if**.

2. I seguenti simboli indicano simboli **non terminali** della grammatica:

   - Lettere maiuscole all'inizio dell'alfabeto:

   $$A, B, C, \ldots, A', A'', \ldots, A_1, A_2, \ldots$$

   - La lettera S che in genere indica il simbolo iniziale della grammatica (a meno che non sia specificato diversamente)
   - Stringhe in minuscolo e in corsivo come ad esempio *expr* o *stmt*.
   - Stringhe qualsiasi tra $<$ e $>$ come ad esempio $<$ OP $>$ oppure $<$ AB $>$.

3. Le lettere maiuscole alla fine dell'alfabeto:

   $$X, Y, Z, \ldots, X', X'', \ldots, X_1, X_2, \ldots$$

   rappresentano un simbolo terminale o non terminale, cioè simboli nell'insieme $\Sigma \cup V$.

4. Le lettere minuscole alla fine dell'alfabeto:

   $$u, v, w, x, y, z, \ldots, u', x', \ldots, x_1, v_2, \ldots$$

   rappresentano *stringhe* di soli simboli terminali (anche vuote), cioè elementi di Σ^*.

5. Le lettere minuscole dell'alfabeto greco:

   $$\alpha, \beta, \delta, \gamma, \ldots, \alpha', \alpha'', \ldots, \beta_1, \gamma_2, \ldots$$

   rappresentano stringhe, anche vuote, formate da simboli terminali o non terminali, cioè elementi di $(V \cup \Sigma)^*$. Tramite le convenzioni viste fino ad ora, vediamo che una generica produzione della grammatica può essere indicata con $A \to \alpha$.

6. Se $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k$ sono tutte le produzioni con lo stesso simbolo a sinistra A (le chiamiamo A-produzioni), allora possiamo scrivere equivalentemente $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Le $\alpha_i, i = 1, 2, \ldots, k$ sono chiamate *alternative* per A.

7. A meno che non si specifichi diversamente, assumeremo che la parte sinistra della prima produzione data per una grammatica sia il simbolo iniziale.

3.2 Alberi di derivazione

Il modo più semplice di generare una stringa da una grammatica è quello di costruire un albero di derivazione.

Definizione 3.2 (Albero di derivazione) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Un albero di derivazione di G è un albero in cui i nodi sono etichettati con i simboli della grammatica $\Sigma \cup V$ e sono rispettate le seguenti proprietà:*

- *La radice dell'albero è etichettata con il simbolo iniziale S*
- *Ogni foglia dell'albero è etichettata con un simbolo terminale*
- *Ogni nodo interno dell'albero è etichettato con un simbolo non terminale A i cui figli, presi da sinistra a destra, sono etichettati con i simboli $X_1 X_2 \cdots X_n$ della parte destra di una qualche produzione $A \to X_1 X_2 \cdots X_n$ in P.*

La stringa w che si ottiene concatenando i simboli associati alle foglie di un albero di derivazione, andando da sinistra verso destra, si chiama stringa associata *all'albero di derivazione. Diciamo anche che un albero è un albero di derivazione per w se w è la sua stringa associata.*

Nel contesto dell'analisi sintattica l'albero di derivazione viene anche chiamato parse tree.

Esempio 3.3 *Consideriamo la seguente grammatica:*

$$E \to E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$$

I simboli terminali sono $(,), +, *, -, \mathbf{id}$. L'unico simbolo non terminali è E che è, ovviamente, anche il simbolo iniziale.

L'albero in Figura 3.1 è un albero di derivazione la cui stringa associata è $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

Tramite gli alberi di derivazione e la nozione di stringa associata possiamo definire precisamente cosa si intende per linguaggio generato da una grammatica.

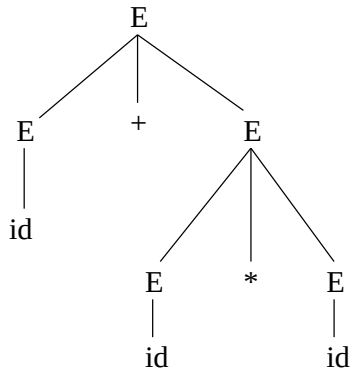


Figura 3.1: Un albero di derivazione per **id** + **id** ∗ **id**.

Definizione 3.4 (Linguaggio generato) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Il linguaggio generato da G è indicato con $L(G)$ ed è l'insieme delle stringhe di $w \in \Sigma^*$ tali che esiste un albero di derivazione di G la cui stringa associata è w.*

3.3 Derivazioni

Definiamo ora il concetto di *derivazione* di una stringa a partire da un simbolo di categoria sintattica. Questa nozione è un'alternativa a quella degli alberi di derivazione. Anch'essa può essere usata per definire il linguaggio delle stringhe generate da una grammatica.

Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Sia α una stringa che contiene almeno un simbolo non terminale A, cioè $\alpha = \delta A \gamma$. Possiamo applicare ad α un *passo di derivazione* utilizzando una produzione di P che ha A come testa. Il passo consiste nel riscrivere α sostituendo al simbolo di categoria sintattica A che abbiamo messo in evidenza il corpo della produzione scelta.

Definizione 3.5 (Passo di Derivazione) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto e sia α una stringa che contiene almeno un simbolo non terminale A. Se $A \to X_1 X_2 \cdots X_k \in P$, allora possiamo riscrivere:*

$$\alpha = \delta A \gamma \Rightarrow_G \delta X_1 X_2 \cdots X_k \gamma$$

Il simbolo $\Rightarrow_G$ rappresenta un passo di derivazione *per la grammatica G. Spesso, se la grammatica che consideriamo è chiara dal contesto, ometteremo il pedice G.*

Definizione 3.6 (Derivazione) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Una derivazione di una stringa $w \in \Sigma^*$ a partire da S è una sequenza:*

$$S = \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \cdots \Rightarrow_G \alpha_n = w$$

dove, per ogni $i \in \{0, 1, \ldots, n-1\}$, $\alpha_i \Rightarrow_G \alpha_{i+1}$ *è un passo di derivazione che riscrive un qualche simbolo non terminale di* α_i.

Si noti che se $i < n$ *allora* $\alpha_i \in (\Sigma \cup V)^*$. *Una stringa di questo tipo, generata cioè con un certo numero di passi di derivazione a partire dal simbolo iniziale, viene chiamata* forma sentenziale.

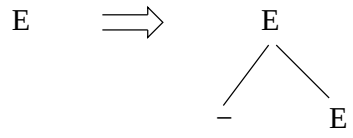
Una derivazione lunga 0 passi è la sequenza composta solo dal simbolo S.

Per indicare una derivazione di zero o più passi da S *ad una certa stringa* α *scriviamo* $S \overset{*}{\Rightarrow}_G \alpha$, *mentre* $S \overset{+}{\Rightarrow}_G \alpha$ *indica una derivazione lunga almeno 1 passo da* S *a* α.

Possiamo definire il linguaggio generato dalla grammatica anche con la nozione di derivazione dicendo che $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}$. Questa definizione è equivalente a quella data usando gli alberi di derivazione, nel senso che l'insieme di stringhe definito con le derivazioni è lo stesso di quello definito con gli alberi di derivazione. Per convincersi di questo basta notare che la costruzione di una derivazione induce in maniera naturale la costruzione di un albero di derivazione. Prendiamo ad esempio una derivazione per la stringa $-(\mathbf{id} + \mathbf{id})$ con la grammatica dell'Esempio 3.3:

$$E \Rightarrow\ -E \Rightarrow\ -(E) \Rightarrow\ -(E+E) \Rightarrow\ -(\mathbf{id}+E) \Rightarrow\ -(\mathbf{id}+\mathbf{id})$$

Ogni passo di derivazione si riflette, nella costruzione del corrispondente albero di derivazione, nell'aggiunta dei figli al nodo corrispondente al simbolo non terminale che si sta riscrivendo. Tali figli sono i simboli del corpo della produzione che si sta usando per il passo di derivazione. All'inizio il solo simbolo E, al passo zero di derivazione, corrisponde alla radice dell'albero. Al primo passo viene applicata la produzione $E \to -E$ e quindi vengono aggiunti due figli ($-$ ed E) alla radice:



Tutte le altre trasformazioni sono riportate in Figura 3.2

Si noti che, durante la derivazione di una stringa dal simbolo iniziale della grammatica, ad ogni passo occorre fare due scelte. Supponiamo di avere una forma sentenziale β (cioè $S \overset{*}{\Rightarrow} \beta$) che non sia una stringa di soli terminali. In generale, per fare un passo di derivazione $\beta \Rightarrow \beta'$, bisogna:

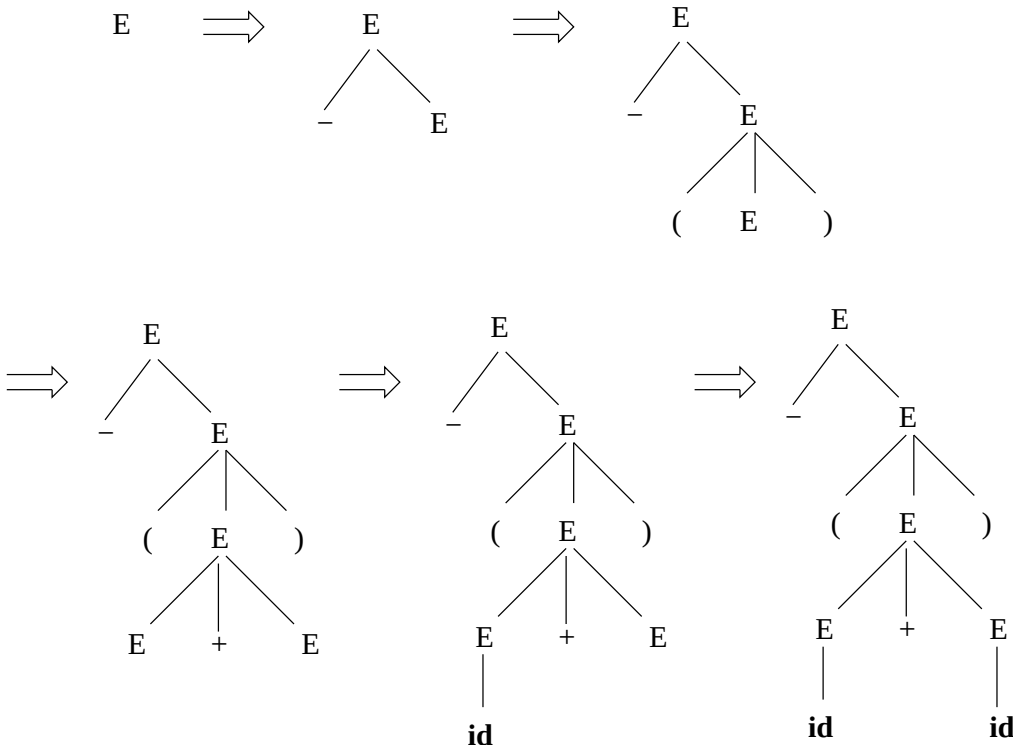


Figura 3.2: Da una derivazione a un albero di derivazione

1. Individuare un simbolo non terminale in β che sarà il candidato per la riscrittura: $\beta = \alpha A \gamma$
2. Scegliere una produzione $A \to \delta$, fra le eventuali possibili scelte per A, con la quale effettuare la riscrittura $\beta = \alpha A \gamma \Rightarrow \alpha \delta \gamma$

Per il primo tipo di scelta possiamo fissare una regola che identifichi univocamente ad ogni passo il simbolo non terminale da riscrivere. Questa restrizione vedremo che sarà molto utile nell'ambito del parsing di una stringa rispetto ad una grammatica. Le due regole principali che si usano sono le seguenti:

- **Derivazione Leftmost** Ad ogni passo si riscrive il simbolo non terminale A di una forma sentenziale β che sta più a sinistra: $\beta = xA\alpha$ (ricordiamo che per x si intende una stringa di soli simboli terminali). Le forme sentenziali che si trovano lungo una derivazione leftmost si chiamano *forme sentenziali sinistre*. Per indicare che ad un passo di derivazione si è applicata la regola leftmost utilizziamo la notazione $xA\alpha \Rightarrow_{lm} x\delta\alpha$[1]. In caso di più passi di derivazione leftmost usiamo $\overset{*}{\Rightarrow}_{lm}$ ($\overset{+}{\Rightarrow}_{lm}$) per zero o più passi (per uno o più passi). Si noti che una derivazione è leftmost se e solo se tutti i passi di cui è composta sono leftmost.
- **Derivazione Rightmost** Ad ogni passo riscriviamo il simbolo non terminale più a destra. Le forme sentenziali sono dette forme sentenziali destre e la notazione usata è $\Rightarrow_{rm}$ con le stesse estensioni viste nel caso leftmost: $S \overset{*}{\Rightarrow}_{rm} \alpha A x \Rightarrow_{rm} \alpha \delta x$.

3.4 Ambiguità

Una questione importante che riguarda le grammatiche libere dal contesto nel loro uso come generatori di linguaggi di cui viene effettuato il parsing è l'ambiguità.

Intuitivamente possiamo vedere la scrittura di una grammatica come la definizione di un algoritmo (ricorsivo) per generare le stringhe di un linguaggio. Questo algoritmo usa le produzioni e costruisce un albero di derivazione (o una derivazione) per una certa stringa data. È facile vedere che durante la generazione di una stringa l'algoritmo (cioè la grammatica) impone certe scelte che riflettono la struttura delle produzioni. La questione dell'ambiguità può essere vista intuitivamente nel seguente modo: la grammatica è ambigua se le produzioni permettono di seguire almeno due strade differenti per generare una certa stringa. Si noti che basta che esista una sola stringa

[1] Omettiamo il simbolo G che indica la grammatica che stiamo usando. In questi casi sarà opportuno sincerarsi che la grammatica G in questione sia chiaramente specificata nel contesto.

per cui questo è vero e tutta la grammatica diventa ambigua. Da questo approccio intuitivo si può anche ricavare una giustificazione del fatto che fare il parsing di grammatiche ambigue risulta molto difficoltoso: il parser che cerca di analizzare una stringa che può essere generata in due (o più) modi non sa decidere quale strada seguire, fra le possibili, nella ricostruzione dell'albero. Inoltre la struttura "troppo libera" delle produzioni rende difficoltosa l'analisi anche delle stringhe che hanno un solo albero di derivazione.

Dopo questa premessa informale definiamo formalmente quando una grammatica è ambigua:

Definizione 3.7 (Ambiguità) *Una grammatica libera dal contesto G si dice ambigua se e solo se esiste una stringa $w \in L(G)$ tale che esistono due alberi di derivazione T e T' di G con le seguenti proprietà:*

- *T e T' sono diversi*
- *T e T hanno w come stringa associata*

Di converso, G non è ambigua se per ogni stringa $w \in L(G)$ esiste uno ed un solo albero di derivazione di G che ha w come stringa associata.

La caratterizzazione dell'ambiguità è stata data usando gli alberi di derivazione. In effetti, visto lo stretto rapporto che c'è tra gli alberi di derivazione e le derivazioni, anche queste ultime possono essere usate per definire l'ambiguità. L'osservazione fondamentale è che se si fissa una regola per scegliere il simbolo da riscrivere ad ogni passo di derivazione allora due derivazioni diverse corrispondono ad alberi di derivazione diversi e due derivazioni uguali corrispondono allo stesso albero di derivazione.

Quindi, ad esempio prendendo la regola leftmost (o rightmost), si ha che una grammatica non è ambigua se e solo se per ogni stringa esiste una unica derivazione leftmost (o rightmost).

Esempio 3.8 *Consideriamo la seguente grammatica:*

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Questa grammatica è ambigua poiché per la stringa $\mathbf{id} + \mathbf{id} * \mathbf{id}$ *esistono i due alberi di derivazione mostrati in Figura 3.3. Equivalentemente esistono due derivazioni leftmost. La seguente corrisponde all'albero in Figura 3.3(a):*

$$E \Rightarrow_{\text{lm}} E + E \Rightarrow_{\text{lm}} \mathbf{id} + E \Rightarrow_{\text{lm}} \mathbf{id} + E * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * \mathbf{id}$$

La seguente invece corrisponde all'albero in Figura 3.3(b):

$$E \Rightarrow_{\text{lm}} E * E \Rightarrow_{E} + E * E \Rightarrow_{\text{lm}} \mathbf{id} + E * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * \mathbf{id}$$

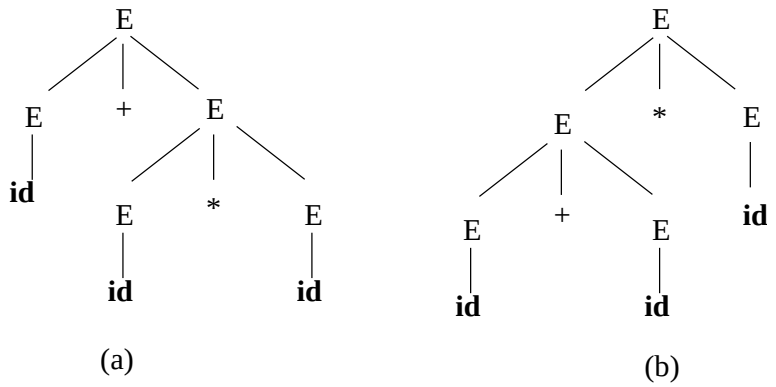


Figura 3.3: Due alberi di derivazione diversi per la stringa $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

Un indizio che indica spesso che una grammatica è ambigua è il fatto che in almeno una produzione è usata la ricorsione "doppia", cioè il simbolo a sinistra della produzione occorre almeno due volte nella parte destra. È indicativo soprattutto se la grammatica genera un linguaggio con operatori binari come quello dell'esempio precedente.

In generale è molto difficile dimostrare che una grammatica *non* è ambigua poiché bisogna dimostrare l'unicità dell'albero di derivazione per *tutte* le stringhe del linguaggio. Dimostrare invece la non ambiguità è semplice: basta esibire un controesempio, cioè una stringa per cui esistono due alberi di derivazione oppure due derivazioni leftmost (o rightmost). Per capire se una grammatica è ambigua oppure no è bene innanzitutto cercare di capire il linguaggio generato dalla stessa e poi capire *come* vengono generate le stringhe: se c'è un algoritmo che impone delle scelte per ogni tipo di stringa generabile oppure se le produzioni lasciano abbastanza libertà tanto da consentire che una certa stringa o un certo gruppo di stringhe possano essere generate seguendo diverse strade.

Nella pratica, se si vuole scrivere una grammatica non ambigua per un certo linguaggio, è utile progettare le produzioni in modo tale che la generazione di ogni stringa del linguaggio debba seguire una ed una sola strada.

Se una grammatica risulta essere ambigua e deve essere usata per generare un parser è bene che sia disambiguata. Infatti molti metodi per generare parser (tra cui tutti quelli che vedremo noi) falliscono se la grammatica che viene considerata è ambigua.

La disambiguazione di una grammatica consiste nella riscrittura delle sue produzioni (anche introducendo o togliendo simboli non terminali) in modo tale da lasciare inalterato il linguaggio generato e da avere un solo albero di derivazione per tutte le stringhe, anche quelle per le quali esistevano più alberi di derivazione associati nella grammatica di partenza.

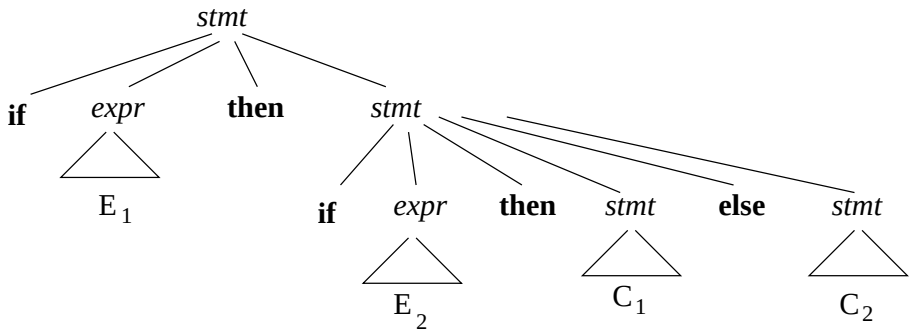


Figura 3.4: Un albero di derivazione per la stringa

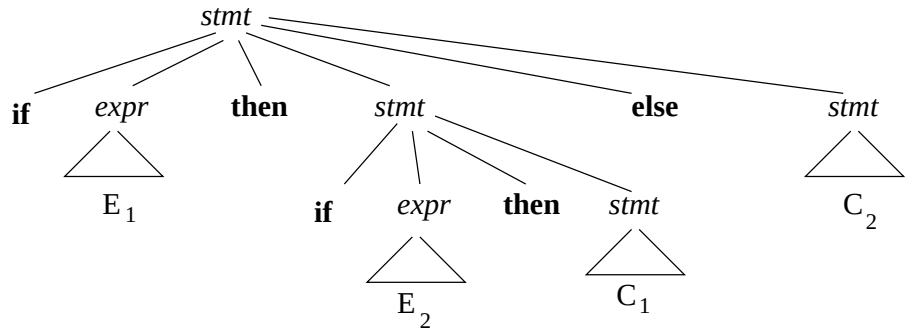


Figura 3.5: Un altro albero di derivazione per la stessa stringa

3.4.1 Un costrutto ambiguo

In questa sezione consideriamo un esempio classico di ambiguità e successiva disambiguazione. Il costrutto che produce il problema è uno dei più usati in tutti i linguaggi di programmazione: il condizionale if-then-else.

L'ambiguità nasce dal fatto che un costrutto condizionale può avere un solo ramo oppure può essere completo e avere due rami. Prendiamo il seguente spezzone di grammatica del Pascal:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
 & | & \textbf{altri\_comandi}
\end{array}
$$

Consideriamo il comando **if** E_1 **then if** E_2 **then** C_1 **else** C_2 dove supponiamo che E_1, E_2, C_1 e C_2 corrispondono a costrutti corretti (espressioni e comandi). Possiamo individuare due alberi di derivazione per il comando dato. Ciò è dovuto al fatto che la grammatica permette di specificare comandi condizionali con e senza **else** senza nessun vincolo.

Nelle Figure 3.4 e 3.5 sono mostrati due alberi di derivazione diversi per la stringa data che sono dimostrazione del fatto che la grammatica che abbiamo considerato è ambigua.

Siamo interessati a disambiguare la grammatica. In Pascal e anche in altri linguaggi di programmazione simili si fissa una regola che permette di interpretare le stringhe come quella che abbiamo considerato sopra in una maniera univoca. La regola che si segue dice che ogni **else** deve essere associato al **then** non associato più vicino. Quindi l'albero di derivazione "giusto" è quello di Figura 3.4.

Questa regola di associazione può essere incorporata direttamente nella grammatica disambiguandola. L'idea è quella di dividere i comandi in due tipologie:

1. i comandi "matched", rappresentati dalla categoria sintattica *matched_stmt*, che sono i comandi in cui tutti i **then** hanno un **else** associato oppure non sono comandi condizionali (generati dalla categoria sintattica *non_cond*)

2. i comandi "unmatched", rappresentati dalla categoria sintattica *unmatched_stmt*, che sono tutti i comandi condizionali che hanno solo il ramo **then** seguito da qualsiasi comando oppure che hanno anche un ramo **else**, ma in questo caso il comando fra il **then** e l'**else** è un comando "matched" e il comando dopo l'**else** è "unmatched".

In questo modo ogni comando che appare tra un **then** ed un **else** non può finire con un **then** non associato seguito da un altro comando qualsiasi. Ecco la nuova grammatica:

stmt	$\rightarrow$	*matched_stmt* | *unmatched_stmt*
matched_stmt	$\rightarrow$	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	*non_cond*
unmatched_stmt	$\rightarrow$	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

In Figura 3.6 è disegnato l'unico albero di derivazione per la stringa che abbiamo considerato sopra. Questa volta non si può costruire un albero simile a quello di Figura 3.5 perché fra il primo **then** e l'**else** non possiamo mettere un comando "unmatched" come un condizionale senza l'**else**.

3.5 Associatività e precedenza degli operatori

La struttura di un albero di derivazione per una certa stringa è molto importante nel contesto del parsing e dei compilatori in generale. Infatti, se una frase di un linguaggio di programmazione viene strutturata in maniera corretta rispetto alla sua semantica, la traduzione della stessa risulta estremamente semplice.

Per capire meglio questo aspetto facciamo l'esempio classico delle espressioni aritmetiche. Possiamo pensare di usare l'albero di derivazione di una

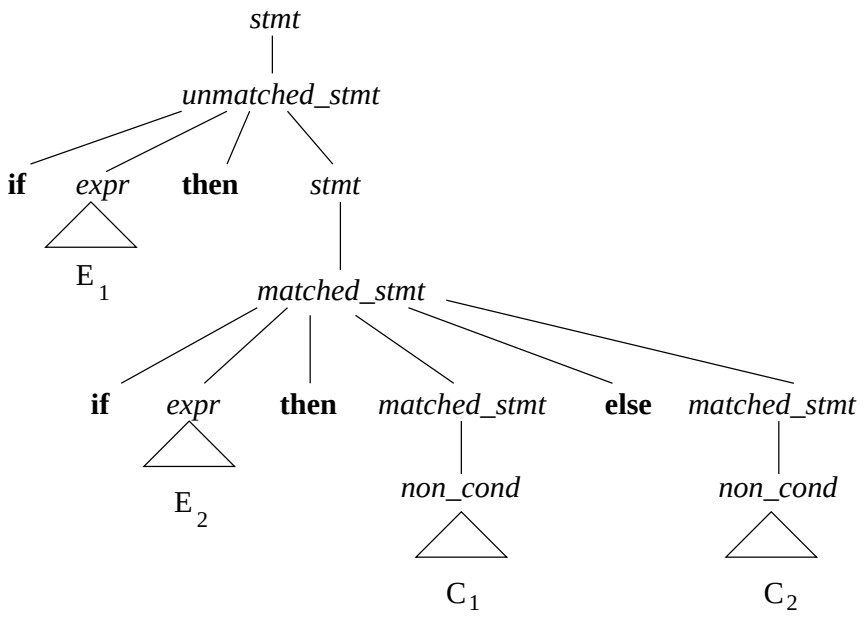


Figura 3.6: L'unico albero di derivazione per la stringa con la nuova grammatica.

certa espressione aritmetica per calcolarne il valore. Si consideri ad esempio l'albero in Figura 3.3(a) per la stringa **id** + **id** $*$ **id**. L'osservazione fondamentale è che ogni nodo interno etichettato E corrisponde ad una sottoespressione il cui valore può essere calcolato in base ai valori associati ai nodi figli. Ad esempio se il nodo ha un unico figlio etichettato **id** allora il valore ad esso associato è il valore associato all'identificatore in memoria (stiamo parlando della semantica di esecuzione del programma). Se il nodo ha tre figli etichettati con $E + E$ rispettivamente allora il suo valore sarà la somma del valore associato al primo figlio E con il valore associato al secondo figlio E (questi valori sono calcolati ricorsivamente). In questo modo il valore associato alla radice è il valore dell'espressione aritmetica.

A questo punto è chiaro perché la struttura dell'albero si rivela fondamentale: deve rispecchiare le regole (semantiche) di valutazione delle espressioni aritmetiche affinché il valore associato alla radice sia quello giusto. Noi conosciamo queste regole: sono regole di associatività e di precedenza. Ad esempio, l'albero di Figura 3.3(b) non può essere usato per valutare l'espressione perché la sua struttura non rispetta la precedenza dell'operatore $*$ di moltiplicazione rispetto all'operatore $+$ di addizione.

In questa sezione vediamo come esprimere l'associatività e la precedenza fra operatori binari (cioè che prendono due argomenti) con le produzioni della grammatica. L'esempio che useremo via via sarà quello delle espressioni aritmetiche con l'addizione, la sottrazione, la moltiplicazione, la divisione e le parentesi tonde fra numeri formati da una sola cifra (per semplificare la

trattazione e non introdurre dettagli inutili).

Innanzitutto chiariamo bene cosa intendiamo per associatività e precedenza.

- Associatività. Prendiamo ad esempio l'operatore $+$. In una espressione come $3+5+7$ non è chiaro a quale $+$ deve essere associato il numero 5 poiché ne ha uno a destra e uno a sinistra ($+$ è un operatore binario e quindi per essere applicato ha bisogno di due operandi: uno alla sua destra e uno alla sua sinistra in notazione infissa). La regola di associatività specifica proprio questo punto: se si stabilisce che il $+$ associa a sinistra allora il 5 viene associato al $+$ alla sua sinistra e la struttura sottintesa dell'espressione di cui sopra è la stessa di $(3 + 5) + 7$ (dove la struttura è stata esplicitata con l'uso delle parentesi). Nel caso di associatività a destra la struttura sarebbe stata $3 + (5 + 7)$.

- Precedenza. Prendiamo due operatori classici con precedenza diversa: $+$ e $*$. In una espressione come $3 + 5 * 7$ ancora una volta non è chiaro a quale operatore deve essere associato il 5 centrale. La regola di precedenza risolve questo punto ponendo gli operatori su livelli diversi di precedenza. Se, come è di solito, il $*$ ha precedenza maggiore del $+$ allora il 5 viene associato al $*$ e quindi la struttura esplicitata dell'espressione è $3 + (5 * 7)$.

Per scrivere una grammatica le cui produzioni riflettano le scelte di precedenza e associatività bisogna innanzitutto definire i vari livelli di precedenza. In ogni livello di precedenza bisogna inserire uno o più operatori che hanno la stessa precedenza (ad esempio il $+$ ed il $-$ nelle espressioni aritmetiche) e per ogni livello si indica se l'associatività deve essere a destra o a sinistra. Per ogni livello definito si crea un simbolo non terminale della grammatica. Illustriamo il procedimento di costruzione delle produzioni con un esempio. Prendiamo i seguenti livelli di precedenza fra gli operatori aritmetici:

1. Fattori (simbolo F), cioè gli operandi. Hanno il maggiore livello di precedenza per definizione.

2. Termini (simbolo T), cioè applicazione di $*$ o $/$. Hanno la stessa precedenza, inferiore a quella dei fattori. L'associatività è a sinistra (come è di solito nei linguaggi)

3. Espressioni (simbolo E), cioè applicazione di $+$ e $-$. Hanno la precedenza più bassa di tutti. L'associatività è a sinistra.

Per il livello più alto si scrivono semplicemente le produzioni:

$$F \to 0 \mid 1 \mid \cdots \mid 9 \mid (E)$$

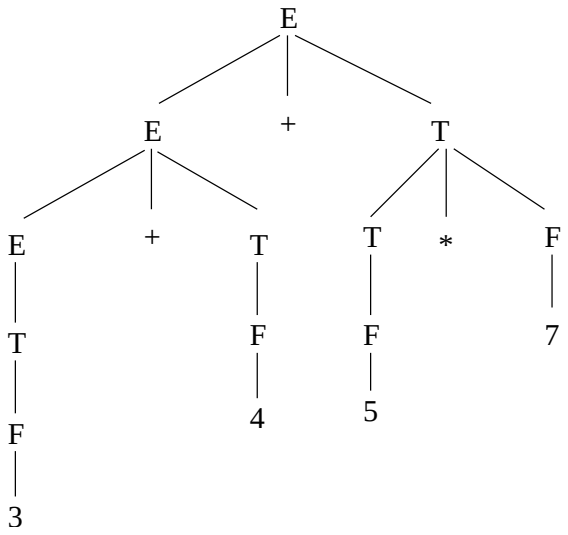


Figura 3.7: Un albero di derivazione per l'espressione $3 + 4 + 5 * 7$.

Si noti che una espressione tra parentesi è da considerarsi come un operando poiché le parentesi forzano l'interpretazione di una espressione permettendo di eludere le regole di precedenza e associatività se occorre.

Nel livello successivo l'associatività a sinistra è espressa mediante la ricorsione a sinistra nelle produzioni. Una produzione ricorsiva a sinistra provoca infatti l'addossamento a sinistra degli alberi di derivazione che la usano e questa struttura corrisponde proprio alla regola semantica di calcolo con l'associatività a sinistra. Le produzioni sono:

$$T \to T * F \mid T/F \mid F$$

Si noti che un fattore può essere sempre visto come un termine (produzione $T \to F$).

Per il successivo e ultimo livello si ha:

$$E \to E + T \mid E - T \mid T$$

In generale ogni elemento di un livello può essere visto come elemento del livello successivo (in questo caso particolare si guardi la produzione $E \to T$).

Attraverso l'imposizione delle regole di precedenza e di associatività abbiamo ottenuto una grammatica non ambigua per le espressioni aritmetiche. In Figura 3.7 è mostrato un albero di derivazione per l'espressione $3+4+5*7$.

3.6 Tecniche di base per la scrittura di grammatiche

Il formalismo delle grammatiche libere dal contesto mette a disposizione pochi e semplici strumenti per la descrizione dei linguaggi, ma questi meccani-

smi, se ben usati e combinati, permettono di definire quasi tutti i costrutti dei tipici linguaggi di programmazione.

In questa sezione analizziamo da un punto di vista più concreto questi meccanismi e vediamo quali sono le tecniche di base per usarli.

Innanzitutto c'è da notare che se non ci sono definizioni ricorsive (dirette o indirette) nelle produzioni di una grammatica allora subito possiamo dire che il linguaggio generato è un insieme finito. Infatti è grazie alla ricorsione che possiamo generare un numero infinito di parole con un numero finito di produzioni.

Ad esempio la grammatica

$$\begin{array}{lcl} S & \rightarrow & aA|bB \\ A & \rightarrow & ab|c \\ B & \rightarrow & ba|c \end{array}$$

non contiene ricorsione e il linguaggio generato è semplicemente l'insieme $\{aab, ac, bba, bc\}$.

La ricorsione può essere diretta o mutua. La ricorsione è diretta quando in una produzione il simbolo di testa è usato anche all'interno del corpo almeno una volta. Un esempio di ricorsione diretta è quello nella produzione $A \rightarrow aA$ nella seguente grammatica:

$$\begin{array}{lcl} S & \rightarrow & aA|bB \\ A & \rightarrow & aA|c \\ B & \rightarrow & ba|c \end{array}$$

L'introduzione di questa ricorsione ha incrementato il linguaggio generato di un numero infinito di parole. Il linguaggio ora è $\{bba, bc\} \cup \{a^n\, c \mid n > 0\}$.

Il precedente è un esempio classico (chiamato ricorsione semplice a destra o ricorsione di coda) con il quale si possono generare parole del tipo $x\, a^n\, y$ dove n può essere positivo o anche zero e x e y sono delle stringhe prefisso e suffisso. Vediamo tutti i casi:

- Poniamo $n \geq 0, x = \epsilon, y = b$. Le produzioni che possiamo scrivere sono $S \rightarrow aS \mid b$. In questo modo il caso $n = 0$ è generato direttamente con il caso base della ricorsione ($S \rightarrow b$) e il caso ricorsivo genera tutte le altre stringhe con $n > 0$ a terminando con il caso base b.

- Poniamo $n > 0, x = \epsilon, y = b$. Le produzioni che possiamo scrivere sono $S \rightarrow aS \mid ab$. Inserendo nel caso base una a evitiamo di scrivere solo una b e otteniamo che $n > 0$ in tutte le stringhe generate.

- Poniamo $n > 0, x = b, y = \epsilon$. Questa volta dobbiamo inserire, all'inizio delle stringhe, qualcosa di diverso dalle a. Ci sono due modi per farlo:

  1. Utilizzare una ricorsione a sinistra (cioè il simbolo di testa si trova all'inizio del corpo della produzione): $S \rightarrow Sa \mid ba$. Il caso $n > 0$ ci ha obbligato a inserire una a nel caso base.

2. Utilizzare un simbolo non terminale in più per inserire il prefisso e poi generare, con ricorsione a destra, le a: $S \to bA$, $A \to aA \mid a$. Nel caso $n \geq 0$ qui metteremmo, come caso base della ricorsione destra su A, la stringa vuota: $S \to bA$, $A \to aA \mid \epsilon$.

- Poniamo $n \geq 0, x = b, y = c$. Nel caso di un prefisso e di un suffisso siamo obbligati a usare un simbolo in più per il prefisso: $S \to bA$, $A \to aA \mid c$. Anche qui, per $n > 0$, avremmo messo come caso base ac.

Un'altra applicazione molto usata della ricorsione diretta è quella che potremmo chiamare "centrale" e che mette il simbolo che ricorre all'interno di due stringhe di terminali, in modo da generare, nelle chiamate ricorsive, un numero uguale di simboli a destra e a sinistra. L'esempio classico è: $S \to aSb \mid ab$ che genera il linguaggio $\{a^n\, b^n \mid n > 0\}$. Il caso base, in questo tipo di applicazione, rappresenta la stringa che va a finire all'interno dei simboli generati in numero uguale.

Facciamo un altro esempio: una grammatica per il linguaggio $\{a^{2n}\, bb\, c^n \mid n \geq 0\}$ è $S \to aaSc \mid bb$. La ricorsione "centrale" è usata per generare eventuali copie di aa e c in numero uguale e il caso base bb può venire usato subito ($n = 0$) o dopo alcune applicazioni della prima produzione ($n > 0$).

Ovviamente è possibile usare insieme e annidare i due tipi principali di ricorsione diretta che abbiamo visto. Ad esempio nella grammatica

$$\begin{aligned} S &\to aSb \mid cB \\ B &\to bB \mid dd \end{aligned}$$

viene annidata una ricorsione destra (con un prefisso c e un suffisso dd) all'interno di una ricorsione "centrale". Il risultato è il linguaggio $\{a^n\, c\, b^m\, dd\, b^n \mid n, m \geq 0\}$. Si noti che le b generate dalla ricorsione "centrale" sono in uguale numero delle a, mentre quelle interne hanno un numero di occorrenze indipendente.

Le varianti di questo schema possono riguardare i casi base e i prefissi/suffissi. Ad esempio, volendo specificare lo stesso linguaggio, ma con $n > 0$, saremmo costretti a inserire sia una a che una b anche nella seconda produzione in questo modo:

$$\begin{aligned} S &\to aSb \mid acBb \\ B &\to bB \mid dd \end{aligned}$$

Il caso $m > 0$, invece ci avrebbe portato a

$$\begin{aligned} S &\to aSb \mid cB \\ B &\to bB \mid bdd \end{aligned}$$

Un altro caso interessante è quello in cui le b interne non hanno suffisso:

$$\begin{aligned} S &\to aSb \mid cB \\ B &\to bB \mid \epsilon \end{aligned}$$

Ora il linguaggio si può scrivere anche come $\{a^n\, c\, b^m \mid m \geq n > 0\}$ poiché le b interne si sommano a quelle alla fine delle stringhe. Dato che le b finali sono sempre in egual numero alle a iniziali si ha che il vincolo $m \geq n$ esprime esattamente la situazione. Se il caso base di B fosse b invece che ϵ allora avremmo che c'è sempre almeno una b interna e, di nuovo, potremmo esprimere la cosa con il vincolo $m > n$.

La mutua ricorsione è meno usata della ricorsione diretta poiché è meno chiara e può indurre facilmente in errore. Vediamone solo un esempio:

$$\begin{aligned} S &\to aA \\ A &\to bB \\ B &\to bA \mid c \end{aligned}$$

Come si vede non c'è ricorsione diretta in nessuna produzione, ma la produzione di A chiama B e una produzione di B chiama A. Il caso base è solo nelle produzioni di B. In questi casi si può semplificare trasformando la grammatica in una equivalente con ricorsione diretta. Basta sostituire, a posto di A, la parte destra della sua unica produzione:

$$\begin{aligned} S &\to abB \\ B &\to bbB \mid c \end{aligned}$$

Il linguaggio è $\{a\, b^{2n+1}\, c \mid n \geq 0\}$.

Per concludere ricordiamo solamente che di fronte alla richiesta di dare una grammatica per un certo linguaggio è bene cercare sempre di dividere le stringhe del linguaggio in diversi casi, ognuno riconducibile a uno schema noto: ricorsione destra, sinistra o "centrale".

Facciamo un esempio finale. Supponiamo di voler dare una grammatica per il seguente linguaggio:

$$L = \{a^n\, b\, c^m\, d\, a^m \mid n \geq 0, m > 0\} \cup \{b^k\, a^{2n} \mid k, m \geq 0\}$$

Innanzitutto possiamo notare che il linguaggio è già naturalmente diviso in due casi dall'unione. Il primo caso è riconducibile a una ricorsione destra con suffisso ($a^n\, b$) seguita da una ricorsione centrale ($c^m\, da^m$) e il secondo caso è la semplice giustapposizione di due ricorsioni destre:

$$\begin{aligned} S &\to A \mid BC \\ A &\to aA \mid bcDa \\ D &\to cDa \mid d \\ B &\to bB \mid \epsilon \\ C &\to aaC \mid \epsilon \end{aligned}$$