LINGUAGGI FORMALI E AUTOMI

Alberto Bertoni
Beatrice Palano

(DISPENSE)

Capitolo 1: Linguaggi e Grammatiche

1 Monoide delle parole, linguaggi e operazioni tra linguaggi

Alfabeto, parola, linguaggio

In generale, con *linguaggio* si intende la capacità d'uso e l'uso stesso di un qualunque sistema di simboli adatti a comunicare. Questa accezione è del tutto generale: si parla di linguaggio della musica o del cinema, così come di linguaggi di programmazione.

Un esempio importante è il *linguaggio naturale*, scritto o parlato. Va osservata la profonda differenza tra il linguaggio *parlato* e quello *scritto*: nel primo caso, il messaggio è veicolato da un segnale acustico continuo, in cui è difficile rilevare la separazione tra lettere o parole, mentre nel secondo il messaggio è codificato da una sequenza di caratteri tipografici e spazi bianchi. Un "testo" nel linguaggio scritto può quindi essere visto come una sequenza finita di simboli scelti da un insieme finito prefissato di simboli come {a ,.., z , A , .. , Z , . , ; , : , !, …, - (="spazio")}. Naturalmente non ogni sequenza di simboli è riconosciuta essere un "testo" del linguaggio: tutti riconoscono che la sequenza "mi-illumino-di-immenso" è una frase dell'italiano scritto, mentre "miil-lumi-nodiimmen-so" non lo è. Un importante e difficile obbiettivo della linguistica è quello di dare una rigorosa descrizione di un linguaggio scritto, specificando quali frasi sono formate correttamente.

Di particolare interesse sono i *linguaggi artificiali*, importanti strumenti per la comunicazione uomo-macchina. Ad esempio, il linguaggio C è descritto dalle sequenze di caratteri che un compilatore C accetta per tale linguaggio. Facendo astrazione dagli esempi precedenti, possiamo ora introdurre la nozione di *linguaggio formale*.

Si considera per prima cosa un arbitrario insieme finito $\Sigma = \{a_1, a_2 \dots a_n\}$, detto *alfabeto*, i cui elementi sono detti *simboli*.

Una *parola* (o *stringa*) su un alfabeto Σ è una sequenza finita di simboli appartenenti a Σ; ad esempio, *aababbab* e *bbababb* sono due parole sull'alfabeto {*a,b*}. E' conveniente considerare la parola non contenente alcun simbolo: essa sarà detta *parola vuota* e indicata con il simbolo ε.

Data una parola w, la *lunghezza* di w (denotata con l(w) oppure |w|) è il numero di simboli (distinti per posizione) che compongono w. La lunghezza della parola vuota è 0.

Esempio 1.1

- L'acido desossiribonucleico (DNA) è il mezzo che trasporta il corredo genetico dell'individuo. Esso è primariamente costituito da una sequenza di quattro molecole fondamentali, individuate da basi azotate quali l' adenina (A), la guanina (G), la citosina (C) e la timina (T). In questa approssimazione, il DNA è una parola di circa 10^6 simboli sull'alfabeto {A,C,G,T}.
- Il numero 17 è rappresentato in base binaria dalla parola 10001 di lunghezza 5 sull'alfabeto {0,1}.

Dato un alfabeto Σ, l'insieme di tutte le parole che si possono ottenere da Σ viene indicato come Σ^*, mentre l'insieme di tutte le parole che si possono ottenere da Σ tranne la parola vuota viene indicato come Σ^+ .

Date due parole $v = x_1 \dots x_n$ e $w = y_1 \dots y_m$, si dice *prodotto di giustapposizione* di v e w (e si indica come $v \cdot w$) la parola $z = x_1 \dots x_n y_1 \dots y_m$. Si osservi che $l(x \cdot y) = l(x) + l(y)$.

Il prodotto di giustapposizione è una operazione binaria su Σ* , che gode della *proprietà associativa* e in cui la parola vuota ε è l'elemento neutro:
1. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
2. $x \cdot \varepsilon = \varepsilon \cdot x = x$

Grazie a queste due proprietà, l'insieme Σ* con l'operazione di giustapposizione e l'elemento neutro ε forma un *monoide.* Più precisamente, la terna $(\Sigma^*, \cdot, \varepsilon)$ è detta *monoide libero* generato da Σ.
Si osservi che il prodotto di giustapposizione non è commutativo: do·lor ≠ lor·do.

Date le parole x, y diremo che x è *prefisso* di y se $y=x \cdot z$ per qualche z, x è *suffisso* di y se y=zx per qualche z, x è *fattore* di y se $y=z \cdot x \cdot w$ per qualche z e w. Ad esempio, si consideri parola "incalzare", "in" è un prefisso, "are" un suffisso" e "calza" un fattore di tale parola.

Un *linguaggio* L sull'alfabeto Σ è un insieme di parole di Σ*, cioè un qualunque sottoinsieme (finito o infinito) $L \subseteq \Sigma^*$. Il linguaggio non contenente alcuna parola viene detto *linguaggio vuoto* e indicato con Ø. Si osservi che Ø è diverso dal linguaggio { ε } contenente solo la parola vuota.

Esempio 1.2

- Le *parole dell'italiano scritto* sono le parole sull'alfabeto {a, b, …, z} elencate in un vocabolario della lingua italiana. Esse formano il linguaggio finito {a, abaco, abate, …, zuppa, zuppiera, zuppo}.
- Le *espressioni aritmetiche* contenenti i numeri 0,1, .., 9, le parentesi (,), e le operazioni +, *, formano un linguaggio L sull'alfabeto {0,1, (,), +, *}, così definito:
    1) $0 \in L, 1 \in L, .., 9 \in L$
    2) se $x \in L$ e $y \in L$, allora $(x+y) \in L$ e $(x*y) \in L$
    3) Nient'altro appartiene a L se non parole ottenute dalle regole 1) e2).

  Ad esempio ((7+5)*2) è una parola in L, mentre (17+5(non è una parola in L. Si osservi che L contiene infinite parole.
- Le *espressioni booleane* contenenti le costanti 0, 1, le variabili a, b, le parentesi (,), le operazioni ¬, ∧, ∨, formano un linguaggio L sull'alfabeto {0,1,a,b,(,), ¬, ∧, ∨}, così definito:
    1) $0 \in L, 1 \in L, a \in L, b \in L$
    2) se $x \in L$ e $y \in L$, allora $(x \wedge y) \in L$, $(x \vee y) \in L$, $\neg x \in L$
    3) Nient'altro appartiene a L se non parole ottenute dalle regole 1) e2).

  Ad esempio $\neg (a \wedge \neg b)$ è una parola in L, mentre $a \neg \wedge 0$ non è una parola in L. Si osservi che anche in questo caso L contiene infinite parole.

Operazioni tra linguaggi

I linguaggi sono sottoinsiemi di Σ*, quindi si possono applicare ad essi le usuali operazioni booleane:
1. *Unione*: dati linguaggi L_1 e L_2 , il linguaggio $L_1 \cup L_2$ contiene tutte le parole che appartengono a L_1 oppure a L_2.
2. *Intersezione*: dati linguaggi L_1 e L_2 , il linguaggio $L_1 \cap L_2$ contiene tutte le parole che appartengono sia a L_1 che a L_2.
3. *Complemento*: il linguaggio L^c contiene tutte le parole che non stanno in L.

Si osservi che l'unione e intersezione di linguaggi finiti è ancora un linguaggio finito, mentre il complemento di un linguaggio finito è un linguaggio infinito.

Abbiamo introdotto in Σ^* il prodotto di giustapposizione. Due naturali operazioni indotte sui linguaggi grazie alla giustapposizione sono il *prodotto* e la *chiusura di Kleene*:

1. *Prodotto*: dati linguaggi L_1 e L_2 , il loro prodotto è il linguaggio $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ e } y \in L_2\}$. Poiché il prodotto è associativo, possiamo definire la potenza L^k, dove $L^0 = \{ \varepsilon \}$ e $L^{k+1} = L^k \cdot L$.
2. *Chiusura di Kleene*: dato un linguaggio L, la sua chiusura è il linguaggio

$$L^* = L^0 \cup L^1 \cup \ldots \cup L^k \cup \ldots = \bigcup_{k=0}^{\infty} L^k$$

dove $L^0 = \varepsilon$ e L^n è$= L \cdot L^{n-1}$, cioè il prodotto di L per sé stesso n volte.

Poniamo ulteriormente $L^+ = \bigcup_{k=1}^{\infty} L^k$

In sostanza L^n è il linguaggio ottenuto moltiplicando in tutti i modi possibili n parole di L, mentre L^* è il linguaggio formato dalle parole ottenute moltiplicando in tutti i modi possibili parole di L, unitamente alla parola vuota ε .
Si osservi che L^* è un sottomonoide di Σ^*, ed in particolare è il più piccolo fra i sottomonoidi di Σ^* che contengono L.

Esempio 1.3
Consideriamo i seguenti linguaggi sull'alfabeto unario $\{a\}$:
$L_1 = \{\varepsilon, a, a^2, a^3\}$e $L_2 = \{a^3, a^4, a^5\}$, dove con a^k intendiamo la parola costituita da k ripetizioni del simbolo a. Allora:

- $L_1 \cup L_2 = \{\varepsilon, a, a^2, a^3, a^4, a^5\}$;
- $L_1 \cap L_2 = \{a^3\}$;
- $L_1^c = \{a^4, a^5, a^6, \ldots\}$ e $L_2^c = \{\varepsilon, a, a^2, a^6,\ldots\}$;
- $L_1 \cdot L_2 = \{a^3, a^4, a^5, a^6, a^7, a^8\} = L_2 \cdot L_1$ (si noti che in generale il prodotto di linguaggi su un alfabeto unario è commutativo, mentre non è commutativo su alfabeti di cardinalità maggiore di 1);
- $L_1^* = \{a\}^*$ e $L_2^* = \{\varepsilon, a^3, a^4, a^5, a^6,\ldots\}$;
- $L_1^+ = \{a\}^*$ e $L_2^+ = \{a^3, a^4, a^5, a^6,\ldots\}$;

Notiamo che in generale una parola in L^+ può essere ottenuta da prodotti di parole in L in diversi modi. Ad esempio, la parola abaaba $\in$ $\{a, aba, abaab\}^+$ può essere ottenuta in due diversi modi:

$$abaaba = aba \cdot aba = abaab \cdot a.$$

Codici

Un linguaggio L in cui ogni parola in L^+ è decomponibile in un unico modo come prodotto di parole di L è detto *codice.* Un codice L possiede dunque l'importante proprietà di *univoca decifrabilità*: data una parola in L^+ esiste un solo modo di ottenerla come prodotto di parole di L. Ad esempio C={0, 01} è un codice, in quanto se una parola in C^+ contiene il simbolo 0 seguito da un altro simbolo 0, allora l'unico fattore possibile è 0, se invece 0 è seguito dal simbolo 1 allora il fattore è sicuramente 01. Viceversa, L={bab, aba, ab} non è un codice, perché la parola ababab può essere fattorizzata in due modi diversi con parole di L: ababab = ab.ab.ab = aba.bab.

Dato un alfabeto V e un codice $C \subseteq \Sigma^*$, una codifica di V in C è una corrispondenza biunivoca $f:V \rightarrow C$. Si dirà che l'elemento $v \in V$ è codificato da f(v) e si scriverà $v = f(v)$.

Un linguaggio L in cui ogni parola non è prefisso di nessun altra parola è chiaramente un codice, in cui ulteriormente ogni parola $w \in L^+$ può essere decodificata in linea leggendo da destra a sinistra. Tali codici sono detti *prefissi*. Non tutti i codici sono prefissi, ad esempio {0, 01} non lo è. E' facile notare che un linguaggio formato da parole di ugual lunghezza è necessariamente un codice prefisso.

Esempio 1.4
I seguenti sono codici prefissi:

- Data la codifica A = ab, B = b, C = aaa, D = aab dell'alfabeto {A,B,C,D} sul codice prefisso {ab, b, aaa, aab}, la parola bababaaabaab $\in$ {ab, b, aaa, aab}* può essere univocamente decomposta come prodotto di parole in {ab, b, aaa, aab}:

  $$bababaaabaab = b \cdot ab \cdot ab \cdot aaa \cdot b \cdot aab$$

  Tale parola può quindi essere univocamente decodificata come BAACBD.
- Il *codice ASCII* rappresenta i 256 caratteri {…,0,…,9,A,B,…, Z, [, \, …} sul codice prefisso $\{0,1\}^8$ formato da tutte le parole binarie di lunghezza 8, cioè da tutte le sequenze di 8 bit. Ad esempio, in tale codifica il carattere 'A' è rappresentato dalla parola binaria 01000001, il carattere 'B' invece dalla parola 01000010, e così via.

2 Rappresentazione di Linguaggi: riconoscitori e generatori

Un linguaggio L è un insieme di parole su un dato alfabeto. Come è possibile dare una descrizione di L? Due soluzioni sono:

1. Se L è finito, può essere rappresentato estensivamente elencando tutte le sue parole: $L = \{w_1, w_2 \dots w_n\}$.
2. Se L è infinito, potremo rappresentarlo solo intensivamente, attraverso cioè una proprietà P(w), descrivibile con una quantità finita di informazione, che risulta vera su tutte e sole le parole di L: $L=\{w \mid P(w)=vero\}$; in altri termini, P esprime cosa deve soddisfare una parola per appartenere ad L.

Due importanti metodi per rappresentare linguaggi infiniti sono quello *generativo* e quello *riconoscitivo*.

Riconoscitori

Dal punto di vista *riconoscitivo*, un linguaggio $L \subseteq \Sigma^*$ è descritto da un *algoritmo* che, avendo in ingresso una parola $w \in \Sigma^*$, dà in uscita 1 se $w \in L$, 0 se $w \notin L$. L'algoritmo, detto riconoscitore, calcola dunque la funzione caratteristica del linguaggio L, cioè la funzione $\chi_L(w)$ = se $w \in L$ allora 1 altrimenti 0.

Esempio 2.1
Si consideri come riconoscitore il *parser* di un compilatore per il linguaggio C: il parser è un algoritmo che, ricevendo in ingresso una stringa w, stabilisce se w è la codifica ASCII di un programma in C sintatticamente corretto; in tal caso, il compilatore dà in output il codice oggetto del programma w, altrimenti elenca gli opportuni messaggi di errore.

Non tutti il linguaggi ammettono un riconoscitore: linguaggi che ammettono riconoscitori sono detti *ricorsivi* o *decidibili*. In questo corso introdurremo due famiglie di riconoscitori, gli *automi a stati finiti* e gli *automi a pila*, capaci di riconoscere due sottoclassi di linguaggi ricorsivi, detti rispettivamente *linguaggi regolari* e *linguaggi liberi da contesto.*

Per capire perché non tutti i linguaggi ammettano un riconoscitore (e ciò implica che non tutti i problemi sono risolubili da un calcolatore digitale!) è necessario precisare meglio il concetto di algoritmo.

Procedure e algoritmi

Supponiamo qui un minimo di familiarità con un linguaggio di programmazione procedurale, per esempio il C. Un programma in C ha due connotazioni. La prima è sintattica: potremo denotare un programma con la parola $w \in \{0,1\}^*$ che corrisponde alla sua codifica ASCII.
La seconda connotazione è semantica: potremo interpretare il programma come un *procedura* che, opportunamente inizializzata, genera una sequenza di passi di calcolo che può terminare dando un risultato.

Per semplicità, ci limitiamo qui al frammento del C formato da procedure che accettano in ingresso parole in $\{0,1\}^*$ e danno come risultato un bit (0 oppure 1). Se w è la parola binaria che corrisponde al codice ASCII di una procedura in C e x è una parola binaria posta in ingresso alla procedura, denotiamo con $F_w(x)$ il risultato dell'esecuzione della procedura w su ingresso x. Se la sequenza di calcolo non termina scriveremo $F_w(x) \uparrow$, se termina scriveremo $F_w(x)\downarrow$. Possiamo avere 3 casi:

$F_w(x) = 0$: la procedura w su ingresso x dà in uscita 0
$F_w(x) = 1$: la procedura w su ingresso x dà in uscita 1
$F_w(x) \uparrow$: la procedura w su ingresso x genera una computazione che non termina.

Un *algoritmo* è semplicemente una procedura w che su qualsiasi ingresso x genera una computazione che termina (dando come risultato, nel nostro caso, 0 oppure 1).

Linguaggi ricorsivi e ricorsivamente numerabili

Sulla base delle nozioni ora introdotte, possiamo proporre la seguente classificazione dei linguaggi:

Definizione 2.1 Un linguaggio L è detto *ricorsivo* (o *decidibile*) se esiste un algoritmo w per cui $F_w(x)=1$ per ogni $x \in L$, mentre $F_w(x)=0$ per ogni $x \notin L$. Un linguaggio L è detto invece *ricorsivamente numerabile* (o *semidecidibile*) se esiste una procedura w per cui $F_w(x)=1$ per ogni $x \in L$, mentre $F_w(x)\uparrow$ per ogni $x \notin L$.

Che cosa vuol dire “linguaggio ricorsivo”? E’ significativa la differenza tra la nozione di linguaggio ricorsivo e quella di linguaggio ricorsivamente numerabile?.

Riguardo alla prima questione, vanno evidenziati due punti. Per prima cosa, se L è un linguaggio ricorsivo, esso può essere specificato con una quantità finita di informazione; essa è data dal testo w del programma che calcola la funzione caratteristica di L. Inoltre, dato $x\in\{0,1\}^*$, risulta possibile decidere se $x\in L$ semplicemente eseguendo il programma w su ingresso x e recuperando il risultato dopo un numero finito di passi di calcolo: se in qualche processo di decisione mi serve il bit della questione “è $x\in L$?”, il programma w attraverso la sua esecuzione genera la conoscenza richiesta.

Se L è invece un linguaggio ricorsivamente numerabile, come nel caso precedente esso può essere specificato con una quantità finita di informazione, data dal testo w del programma che risponde 1 quando $x\in L$ e non termina l’esecuzione quando $x\notin L$. Contrariamente a prima, tuttavia, quando $x\notin L$ non è possibile recuperare questa informazione in un numero finito di passi di calcolo: se in qualche processo di decisione mi serve il bit della questione “è $x\in L$?”, il programma w attraverso la sua esecuzione genera la conoscenza richiesta solo quando $x\in L$!.

E’ chiaro che se un linguaggio L è ricorsivo, allora L è anche ricorsivamente numerabile. Infatti se L è ricorsivo esiste una procedura w che calcola la sua funzione caratteristica; costruiamo ora una nuova procedura che prima simula w poi, se l’uscita è 0, genera una computazione che non termina. Questo prova che L è anche ricorsivamente numerabile.

Il viceversa non è ovvio, perché non è chiaro come sostituire una computazione che non termina con una che termina e dà come risultato 0. Infatti non è possibile.
Per capire perché esistono linguaggi ricorsivamente numerabili che non sono ricorsivi, dobbiamo richiamiare il concetto di interprete. Un interprete per il nostro frammento del C è un programma u (che per semplicità supporremo scritto in C) che accetta in ingresso parole in $\{0,1\}^*\$\{0,1\}^*$ e che, avendo come ingresso la parola x$w, simula l’esecuzione della procedura codificata con w su ingresso x:

$$F_u(x\$w)= F_w(x)$$

Teorema 2.1 Sia $A = \{ x : F_u(x\$x)\downarrow \}$. Allora A è ricorsivamente numerabile ma A non è ricorsivo.
Dimostrazione: Considera per prima cosa la seguente procedura

```
Procedura RICNUM (x∈{0,1}*)
        (1) Calcola y=x$x
        (2) Simula l'interprete u su ingresso y
        (3) Return(1)
```

E’ chiaro che se la precedente procedura è inizializzata con $x\in A$, allora viene generata una computazione che termina e dà in uscita 1, altrimenti viene generata una computazione che non termina. Questo prova che A è ricorsivamente numerabile.
Dimostriamo ora per assurdo che A non è ricorsivo, usando una tecnica introdotta da Cantor, detta tecnica di diagonalizzazione. Supponiamo per assurdo che A sia ricorsivo. Allora possiamo realizzare la seguente procedura:

```
Procedura ASS(x∈{0,1}*)
   (1) if  x∈A   then return(1- Fu(x$x) )
                 else return (0)
```

Mostriamo che questo porta ad un assurdo. Sia e il codice ASCII della procedura ASS. Abbiamo due casi:

1. $e \in A$. In questo caso l'esecuzione della procedura e su ingresso e dà come risultato 1- $F_u(e\$e)$. Ma il risultato dell'esecuzione di e su ingresso e è per definizione $F_e(e)= F_u(e\$e)$. Si avrebbe allora $F_u(e\$e)=1- F_u(e\$e)$, che è assurdo poiché $F_u(e\$e)$ vale o 0 o 1.
2. $e \notin A$. Per definizione di A, se $e \notin A$ allora $F_u(e\$e)\uparrow$. Se $e \notin A$ l'esecuzione della procedura e su ingresso e dà 0. Ma il risultato dell'esecuzione di e su ingresso e è per definizione $F_e(e)= F_u(e\$e)$. Si avrebbe allora l'assurdo che $F_u(e\$e)\uparrow$ ma contemporaneamente $F_u(e\$e)=0$.

□

Il linguaggio A è un esempio di linguaggio non ricorsivo, e tuttavia è ricorsivamente numerabile. Mostriamo ora che A^c non è neanche ricorsivamente numerabile.

Teorema 2.2 Sia $A = \{ x : F_u(x\$x)\downarrow \}$. Allora A^c non è ricorsivamente numerabile.
Dimostrazione: Sappiamo che A è ricorsivamente numerabile, quindi esiste una procedura w tale che $F_w(x)=1$ per ogni $x \in A$, mentre $F_w(x)\uparrow$ per ogni $x \notin A$. Se per assurdo anche A^c fosse ricorsivamente numerabile, esisterebbe una procedura z tale che $F_z(x)=1$ per ogni $x \notin A$, mentre $F_z(x)\uparrow$ per ogni $x \in A$. Considera la seguente procedura:

Procedura ASS($x \in \{0,1\}^*$)
(1) k=0
(2) while [sia w che z su ingresso x non terminano al passo k] do k=k+1
(3) if [w su ingresso x termina al passo k] then return(1) else return(0)

La procedura precedente termina sempre e calcola la funzione caratteristica di A. Ma allora A sarebbe ricorsivo, mentre da Teorema 2.1 sappiamo che A non è ricorsivo.

□

Sistemi generativi

Un'altra modo per rappresentare linguaggi è quello *generativo*; introduciamo questa nozione in relazione al concetto di *sistema formale* o *calcolo logico*. In particolare in questo corso introdurremo un importante modello di sistema generativa: la *grammatica*.

Motiviamo il concetto di *calcolo logico* a partire da una problematica che in qualche misura tutti conoscono: la geometria elementare nel piano. In una trattazione formale della geometria, si può seguire questo approccio:

1. Vengono per prima cosa introdotte vari nomi (piano, punto, linea retta, ecc.) mediante le quali sarà possibile comporre un insieme infinito di affermazioni, rappresentabili da un linguaggio L= {f: f affermazione sulla geometria elementare}. Esempi di tali affermazioni sono le seguenti frasi (viste come parole del linguaggio): le diagonali di un rombo sono perpendicolari, i tre lati di un triangolo sono paralleli, …. .
2. Si interpretano le affermazioni avendo come modello il piano, i suoi punti e le sue rette. Alcune delle affermazioni in L risultano vere (esempio: le diagonali di un rombo sono perpendicolari), altre false (esempio: i tre lati di un triangolo sono paralleli). In linea di principio, è possibile quindi considerare il linguaggio $G \subset L$ formato dalle affermazioni vere: G = {t: t affermazione vera della geometria elementare}.

Un calcolo logico deve permettere di "dimostrare" tutte e sole le affermazioni vere, date nel linguaggio G. Le dimostrazioni dovranno a loro volta poter essere descritte da parole di un linguaggio D, così da dar senso (vero/falso) alla frase:

d è la dimostrazione di f

E' ragionevole richiedere che:

1. Deve essere possibile verificare in un numero finito di passi se d è la dimostrazione di f oppure no.
2. Se d è la dimostrazione di f, allora f deve essere un'affermazione vera nella geometria elementare (correttezza del calcolo: si possono dedurre solo affermazioni vere)
3. Se t è un'affermazione vera della geometria elementare, allora deve esistere una sua dimostrazione d (completezza del calcolo: ogni affermazione vera è dimostrabile)

Generalizzando rispetto all'esempio precedente, definiamo:

Definizione 2.2 Un *calcolo logico* è dato da una funzione $V:\Sigma^*xU^*\rightarrow\{0,1\}$ calcolabile da un algoritmo A. Esiste cioè un algoritmo A che, avendo in ingresso le parole x e d, dà in uscita 1 se V(x,d) = 1, 0 se V(x,d) = 0.

U* denota l'insieme delle potenziali "dimostrazioni". Se V(x,d) = 1 si dirà che "d è una dimostrazione (o testimone) di x". L'esistenza dell'algoritmo A soddisfa la richiesta che sia possibile verificare in un numero finito di passi se d è la dimostrazione di x oppure no.

Dato ora un linguaggio $L\subseteq \Sigma^*$, il calcolo logico V è detto *corretto* per L se V(x,d) = 1 implica $w\in L$. Questa condizione viene detta *correttezza* del calcolo perché, se x ammette una dimostrazione d nel calcolo, allora x deve appartenere a L.

Il calcolo logico V è detto *completo* per L se, ogni qual volta $x\in L$, allora esiste $d\in U^*$ per cui V(x,d)=1. Questa condizione viene detta *completezza* del calcolo perché garantisce che, quando $x\in L$, è sempre possibile trovarne una dimostrazione d nel calcolo.

Definizione 2.3 Dato un linguaggio $L\subseteq \Sigma^*$, un *calcolo logico* per L è un calcolo logico V corretto e completo per L. In tal caso sarà $L=\{x \mid \exists d\ V(x,d)=1\}$.

Calcoli logici e linguaggi ricorsivamente numerabili

Non tutti i linguaggi ammettono un calcolo logico; mostriamo qui che i linguaggi che ammettono calcoli logici sono tutti e soli i linguaggi *ricorsivamente numerabili*.

A tal riguardo, consideriamo per prima cosa una corrispondenza biunivoca tra U* e l'insieme dei numeri naturali $N=\{x \mid 1,2,3,\ldots.\}$. Una corrispondenza naturale è per esempio quella indotta dall'ordine lessicografico, così ottenuta:

1. Si ordinano per prima cosa i simboli di U
2. Date $x,y\in U^*$, sarà $x\le y$ se la lunghezza di x è minore della lunghezza di y oppure, nel caso di ugual lunghezza, x precede y nell'ordine alfabetico.

Le parole in U* risultano ordinate totalmente dalla relazione ≤ . Indicheremo con d_1 la prima parola, d_2 la seconda parola, … , d_n la n-sima parola in U*. Vale:

Teorema 2.3 L ammette un calcolo logico se e solo se L è ricorsivamente numerabile.
Dimostrazione:
($\Rightarrow$) Sia V(x,d) un calcolo logico per $L \subseteq \Sigma^*$. Consideriamo la seguente procedura:

```
Procedura L(x: parola di Σ*)
n = 1
while V(x,d_n) = 0 do n = n+1
return(1)
```

Se $x \in L$, esiste un n per cui $V(x,d_n)=1$; la Procedura L di conseguenza su ingresso x termina e restituisce 1. Se invece $x \notin L$, per ogni n vale che $V(w,d_n)=0$; in questo caso la Procedura L non termina. Ne segue che L è ricorsivamente numerabile.

($\Leftarrow$) Supponiamo che esista una procedura w per cui $F_w(x)=1$ per ogni $x \in L$, mentre $F_w(x)\uparrow$ per ogni $x \notin L$. Costruiamo il seguente calcolo logico:

$V(x,d_n) = 1$, se la procedura w termina in n passi, 0 altrimenti.

Se $x \in L$, esiste un n per cui la procedura w su ingresso x termina in n passi, e quindi esiste la dimostrazione $d = d_n$ tale che $V(x,d) = 1$. Se invece $x \notin L$, la procedura w su ingresso x non termina e quindi per ogni n si ha $V(x,d_n) = 0$.

□

Il risultato precedente apre al seguente scenario. Dato un linguaggio L, si hanno i seguenti due casi:

- L è ricorsivamente numerabile. Esiste allora un calcolo logico per L, corretto e completo.
- L non è ricorsivamente numerabile. Allora qualsiasi calcolo logico V che è corretto per L è necessariamente incompleto. E' quindi possibile trovare una parola x che sta in L, senza che esista una dimostrazione di questo fatto poichè V(x,d)=0 per ogni d.

X Problema di Hilbert

La scoperta di linguaggi di interesse logico non ricorsivamente numerabili è il nucleo dei risultati di incompletezza trovati da Goedel intorno al 1930. Presentiamo qui un esempio basato sul X Problema di Hilbert, uno dei 23 problemi aperti proposti da Hilbert durante il Congresso di Matematica di Parigi nel 1900. Tale problema richiede di trovare tecniche risolutive per le equazioni diofantee (ED), ed in particolare di determinare un metodo per decidere se, dato un polinomio a coefficienti interi $p(x_1, x_1, ..., x_n)$, l'equazione $p(x_1, x_1, ..., x_n) = 0$ ammette una soluzione sugli interi.

Il problema può essere descritto in maniera formale considerando il linguaggio:

$$ED = \{ \exists x_1 \exists x_2, ..., \exists x_n\ p(x_1, x_2, ..., x_n) = 0 \mid p(x_1, x_1, ..., x_n) \text{ polinomio a coefficienti interi} \}.$$

Fissata la struttura Z=<{….,-1,0,1,2,…},+, .> degli interi relativi con somma e prodotto, l'affermazione $\exists x_1 \exists x_2, ..., \exists x_n\ p(x_1, x_2, ..., x_n) = 0$ risulta vera se esistono interi $a_1, a_2, ..., a_n$ tali che $p(a_1, a_2, ..., a_n) = 0$.

Esempio 2.2
L'affermazione $\exists x_1 \exists x_2 (6x_1-10x_2 = 5)$ risulta falsa. Se infatti esistessero due interi a_1,a_2 per cui $6a_1-10a_2=5$, sarebbe $5=2.(3a_1-5a_2)$, il che implica che 5 è pari!
L'affermazione $\exists x_1 \exists x_2 (x_1^4-5 x_2^2=1)$ risulta invece vera, come è *dimostrato* dagli interi 3, 4: $3^4-5.4^2 = 1$

L'insieme delle affermazioni in ED che, interpretate sugli interi, risultano vere forma allora un linguaggio $X \subset ED$ dato da:

$$X = \{ \exists x_1 \exists x_2,..., \exists x_n\ p(x_1, x_2,..., x_n) = 0 \ \mid \text{esistono interi } a_1,a_2,..., a_n \text{ tali che } p(a_1,a_2,..., a_n) = 0\}.$$

Andiamo ora a classificare il linguaggio X. Per prima cosa vale:

Teorema 2.4 X è ricorsivamente numerabile.
Dimostrazione: Poiché l'affermazione $\exists x_1 \exists x_2,..., \exists x_n\ p(x_1, x_2,..., x_n) = 0$ è vera se e solo se si possono trovare interi $a_1,a_2,..., a_n$ tali che $p(a_1,a_2,..., a_n) = 0$, un calcolo logico V per X può essere dato da:

$$V(\exists x_1 \exists x_2,...,\exists x_n\ p(x_1,x_2,...,x_n) = 0\ ,\ a_1,a_2,..., a_n)\ ) = 1, \text{ se } p(a_1,a_2,..., a_n) = 0, \text{ altrimenti } 0.$$

□

Il celebre risultato ottenuto nel 1967 da Matiyasevich (e di cui non possiamo qui dare la dimostrazione) asserisce invece:

Teorema 2.5 (Matiyasevich) X non è ricorsivo.

La non ricorsività di X preclude dunque la possibilità di trovare un algoritmo per la soluzione delle equazioni diofantee, rispondendo in modo negativo al X Problema di Hilbert.

Indaghiamo ora un nuovo problema, che chiameremo Ineguaglianze Diofantee (ID), collegato al precedente. ID chiede di decidere se, dato un polinomio $p(x_1,x_2,...,x_n)$ a coefficienti interi, risulta $p(a_1,a_2,..., a_n) \neq 0$ per tutti gli interi $a_1,a_2,..., a_n$. Esso è descritto dal linguaggio Y dove:

$$Y = \{ \forall x_1 \forall x_2,..., \forall x_n\ \ p(x_1, x_2,..., x_n) \neq 0 \ \mid p(a_1,a_2,..., a_n) \neq 0 \text{ per tutti gli interi } a_1,a_2,..., a_n \}.$$

Il linguaggio Y può essere classificato come segue:

Teorema 2.6 Y non è ricorsivamente numerabile.
Dimostrazione: Si osservi che l'affermazione $\forall x_1 \forall x_2 ... \forall x_n\ \ p(x_1, x_2,..., x_n) \neq 0$ è vera se e solo se l'affermazione $\exists x_1 \exists x_2 ... \exists x_n\ p(x_1, x_2,..., x_n) = 0$ è falsa. Quindi:

$$\forall x_1 \forall x_2 ... \forall x_n\ \ p(x_1, x_2,..., x_n) \neq 0\ \in\ Y \Leftrightarrow \exists x_1 \exists x_2,...,\exists x_n\ p(x_1,x_2,...,x_n) = 0\ \notin\ X$$
$$\Leftrightarrow \exists x_1 \exists x_2,...,\exists x_n\ p(x_1,x_2,...,x_n) = 0\ \in\ X^c$$

Se Y fosse ricorsivamente numerabile anche X^c lo sarebbe. Da **Teorema 2.4** sappiamo che X è ricorsivamente numerabile; essendo sia X che X^c ricorsivamente numerabili, X sarebbe ricorsivo, contro il **Teorema 2.5** (Matiyasevich**).**

□

Per quanto visto, Y non può ammettere nessun calcolo logico che sia corretto e completo. In particolare, preso un qualsiasi calcolo logico V corretto per Y, allora tale calcolo è necessariamente incompleto. Questo significa che esiste una affermazione $\forall x_1 \forall x_2 \dots \forall x_n\ p(x_1, x_2, \dots, x_n) \neq 0$ tale che:

1. L'affermazione $\forall x_1 \forall x_2 \dots \forall x_n\ p(x_1, x_2, \dots, x_n) \neq 0$ è vera, e cioè risulta che $p(a_1, a_2, \dots, a_n) \neq 0$ per tutti gli interi $a_1, a_2, \dots, a_n$
2. La stessa affermazione $\forall x_1 \forall x_2 \dots \forall x_n\ p(x_1, x_2, \dots, x_n) \neq 0$ non ammette tuttavia una dimostrazione in V, cioè vale che $V(\forall x_1 \forall x_2 \dots \forall x_n\ p(x_1, x_2, \dots, x_n) \neq 0, d) = 0$ per ogni dimostrazione d.

□

Problemi più generali, la cui soluzione non è fornita da una semplice risposta booleana, possono essere modellati mediante funzioni da interi in interi. Il dominio di una funzione rappresenta l'insieme delle istanze del problema; le immagini rappresentano le soluzioni. Indichiamo con F una funzione che codifica un particolare problema P' che ha soluzione algoritmica. Un possibile algoritmo per il calcolo di F(x), il cui valore equivale alla soluzione di P' su instanza x, può essere progettato usando un riconoscitore per il linguaggio $L_F = \{a^n b^{F(n)} \mid n \geq 0\}$ (è sufficiente interrogare il riconoscitore su parole della forma $a^x b^y$, con $y \geq 0$, finchè non si ottiene una risposta affermativa).

3 Un sistema generativo: le grammatiche

Consideriamo il seguente insieme di regole descritte informalmente, che permettono la generazione di alcune frasi dell'italiano:

1. Una <Frase> è la giustapposizione di un <Soggetto>, di un <Predicato>, di un <Complemento>.
2. Un <Soggetto> è la giustapposizione di un <Articolo> e di un <Nome>.
3. Un <Complemento> è la giustapposizione di un <Articolo> e di un <Nome>.
4. Un <Articolo> è "il".
5. Un <Nome> è "cane" oppure "gatto" oppure "topo
6. Un <Predicato> è "mangia" oppure "teme"

Esempi di frasi generate sono "il gatto mangia il topo" oppure "il cane teme il gatto". Osserviamo che tali frasi sono parole sull'alfabeto {il, cane, gatto, topo, mangia, teme}, quindi le regole precedenti denotano un linguaggio $L \subseteq$ { il, cane, gatto, topo, mangia, teme}*, mentre i simboli <Frase>, <Soggetto>, <Complemento>, <Predicato>, <Articolo>, <Nome> sono utilizzati per generare tale linguaggio, ma non fan parte di parole del linguaggio: essi vengono quindi detti *metasimboli* o *simboli non terminali*, mentre {il, cane, gatto, topo, mangia, teme} sono detti *simboli terminali* (o semplicemente simboli).

Le regole 1,2,3,4,5,6 possono essere interpretate come regole di produzione come segue:

<Frase> → <Soggetto> <Predicato> <Complemento>
<Soggetto> → <Articolo> <Nome>
<Articolo> → il
<Nome> → cane / gatto / topo
<Predicato> → mangia / teme

Una regola di produzione è data quindi da una coppia $\alpha \rightarrow \beta$, dove α (parte sinistra) è una parola non vuota di simboli (terminali o non) e β, in modo analogo, una parola di simboli (terminali o non); si osservi tuttavia che nel nostro caso la parte sinistra di una regola è sempre costituita da un solo metasimbolo. La applicazione di una regola del tipo $\alpha \rightarrow \beta$ ad una parola $x\alpha y$ produce la parola $x\beta y$, ottenuta sostituendo il fattore α con β; l'applicazione della regola sarà denotata $x\alpha y \Rightarrow x\beta y$.

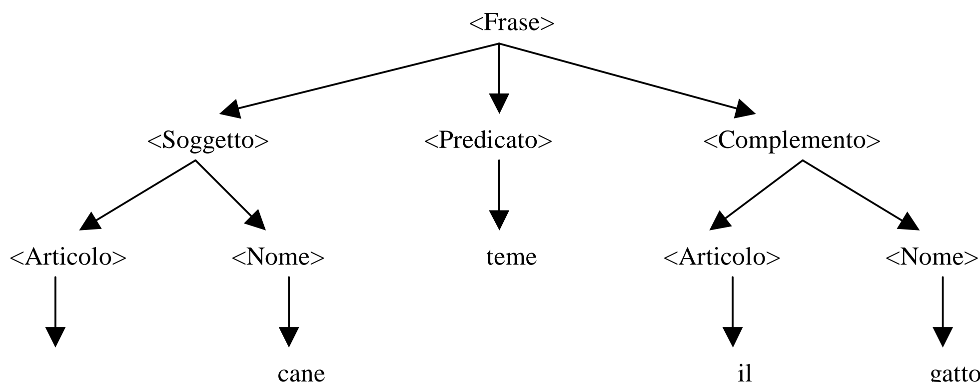
Date due parole w e z, diremo $w \Rightarrow^* z$ se z può essere ottenuto da w applicando un numero finito di regole di produzione; nel nostro caso, ad esempio, possiamo rilevare che:

<Frase>⇒* il cane teme il gatto

Infatti:
<Frase> ⇒ <Soggetto> <Predicato> <Complemento> ⇒
⇒<Articolo> <Nome> <Predicato> <Complemento>⇒ il <Nome> <Predicato> <Complemento>⇒
⇒il cane <Predicato> <Complemento> ⇒il cane teme <Complemento>⇒
⇒il cane teme <Articolo> <Nome>⇒il cane teme il <Nome>⇒ il cane teme il gatto

Si noti che nella derivazione precedente le regole di produzione sono state applicate in ogni parola al primo metasimbolo da sinistra. Il linguaggio generato è formato dalle parole w ∈ { il, cane, gatto, topo, mangia, teme}* tali che <Frase>⇒*w. Il metasimbolo <Frase> gioca quindi il particolare ruolo di "parola scelta inizialmente" e viene chiamato *assioma*.

Una rappresentazione grafica della precedente derivazione è data dal seguente albero:



Astraendo dall'esempio precedente, siamo pronti ad introdurre in generale il concetto di grammatica G e di linguaggio L(G) generato da G.

Definizione 3.1 una grammatica G è una quadrupla $< \Sigma, Q, P, S>$ dove:
1. Σ e Q sono due alfabeti finiti disgiunti, rispettivamente di simboli terminali e metasimboli (o simboli non terminali).
2. *P* è un insieme finito di regole di produzione; una regola di produzione è una coppia $\alpha \rightarrow \beta$ di parole con $\alpha \in (\Sigma \cup Q)^+$ e $\beta \in (\Sigma \cup Q)^*$.
3. S è un elemento in Q, detto *assioma* o *simbolo di partenza*.

Fissata una grammatica G, date due parole $w,z \in (\Sigma \cup Q)^*$ diremo che z è derivabile in G da w in un passo, scrivendo $w \Rightarrow_G z$, se $w = x\alpha y$, $z = x\beta y$ e $\alpha \rightarrow \beta$ è una regola in *P*.
Una *derivazione* di z da w in G è una sequenza $w \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \dots \Rightarrow_G w_m \Rightarrow z$, per cui sia $w \Rightarrow_G w_1$, ..., $w_i \Rightarrow_G w_{i+1}$ .., $w_m \Rightarrow_G z$; diremo infine che z è derivabile in G da w, scrivendo $w \Rightarrow_G^* z$, se w=z oppure se esiste una derivazione di z da w in G.

Il linguaggio L(G) generato dalla grammatica G è l'insieme di parole sull'alfabeto Σ derivabili dall'assioma S, cioè:

$$L(G) = \{w \mid w \in \Sigma^* \text{ e } S \Rightarrow_G^* w\}$$

Due grammatiche G_1 e G_2 sono dette *equivalenti* se generano lo stesso linguaggio, cioè se $L(G_1)=L(G_2)$.

Osserviamo che una derivazione d è una parola in $(\Sigma \cup Q \cup \{\Rightarrow_G\})^*$; consideriamo ora la seguente funzione:

V(w,d) = se d è una derivazione di w da S in G allora 1 altrimenti 0

E' facile disegnare un algoritmo che calcola V, ed inoltre $L=\{w \mid \exists d\ V(w,d)=1\}$. Possiamo concludere che se il linguaggio L è generato da una grammatica G allora L è ricorsivamente numerabile; è possibile mostrare anche la proposizione inversa: se L è ricorsivamente numerabile, allora si può trovare una grammatica G per cui L=L(G). Si può concludere:

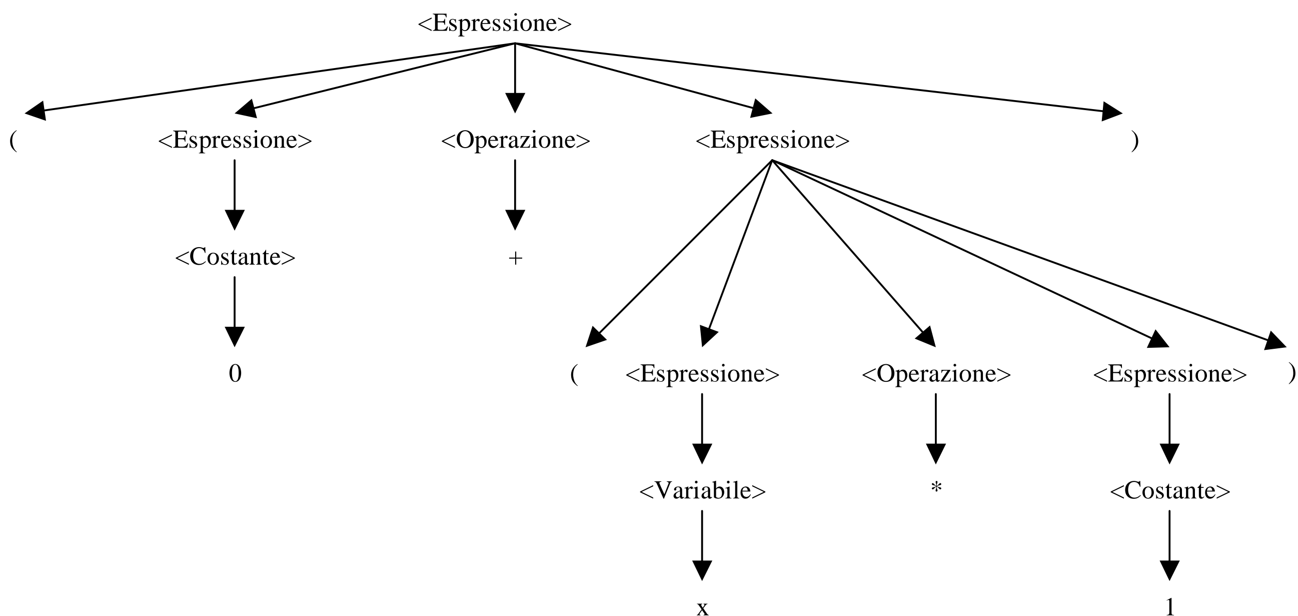
Teorema 3.1 Un linguaggio L è generato da una grammatica se e solo se L è ricorsivamente numerabile.

Il risultato precedente asserisce che se per un linguaggio L esiste un calcolo logico corretto e completo, allora L è generabile da una grammatica. Le grammatiche risultano dunque sistemi formali per esprimere calcoli logici.

Esempio 3.1
Sia L il linguaggio formato dalle espressioni parentesizzate ottenute a partire da variabili x,y, da costanti 0,1, contenenti operazioni * , + . E' chiaramente $L \subset \{x , y , 0 , 1 , * , + . (,)\}^*$; per esempio, $((x+1)*y) \in L$ mentre $x)+01) \notin L$.

L può essere generato dalla seguente grammatica:

1. Alfabeto terminale Σ={x , y , 0 , 1 , . , + . (,) }..
2. Alfabeto non terminale Q={<Espressione>,<Variabile>,<Costante>,<Operazione> }
3. Regole di produzione *P*:
   <Espressione> → <Costante> / <Variabile> / (<espressione> <operazione> <espressione>)
   <Costante> → 0 / 1
   <Variabile> → x / y
   <Operazione> → + / *
4. Assioma S = <Espressione>



Il precedente albero identifica una derivazione della parola (0+(x*1)), dimostrando che tale parola è una espressione parentesizzata "corretta".

Esempio 3.2
Sia L il linguaggio $=\{a^n\ b^n\ c^n \mid n\geq 1\}$.

L può essere generato dalla seguente grammatica:
1. Alfabeto terminale $\Sigma=\{a, b, c\}$
2. Alfabeto non terminale $Q=\{S, B, C\}$
3. Regole di produzione *P*:
   $S \rightarrow aSBC$, $S \rightarrow aBC$, $CB \rightarrow BC$, $aB \rightarrow ab$, $bB \rightarrow bb$, $bC \rightarrow bc$, $cC \rightarrow cc$
4. Assioma: S

La parola $a^n\ b^n\ c^n$ può essere derivata come segue:

$S \Rightarrow_G^* a^{n-1}\ S(BC)^{n-1}$	applicazione iterata della regola $S \rightarrow aSBC$
$a^{n-1}\ S(BC)^{n-1} \Rightarrow_G a^n\ (BC)^n$	applicazione iterata della regola $S \rightarrow aBC$
$a^n\ (BC)^n \Rightarrow_G^* a^n\ B^n\ C^n$	applicazione iterata della regola $CB \rightarrow BC$
$a^n\ B^n\ C^n \Rightarrow_G a^n\ bB^{n-1}\ C^n$	applicazione della regola $aB \rightarrow ab$
$a^n\ bB^{n-1}\ C^n \Rightarrow_G^* a^n\ b^n\ C^n$	applicazione iterata della regola $bB \rightarrow bb$
$a^n\ b^n\ C^n \Rightarrow_G a^n\ b^n\ cC^{n-1}$	applicazione della regola $bC \rightarrow bc$
$a^n\ b^n\ cC^{n-} \Rightarrow_G^* a^n\ b^n\ c^n$	applicazione iterata della regola $cC \rightarrow cc$

Infine lasciamo al lettore il compito di dimostrare che con questa grammatica non possono essere derivate parole che non sono del tipo $a^n\ b^n\ c^n$.

4 Classificazione delle grammatiche e gerarchia di Chomsky

E' possibile classificare le grammatiche in funzione del tipo di regola di produzione. Una interessante classificazione è quella proposta da Chomsky in base a considerazioni di carattere linguistico e di semplicità di trattazione matematica. Viene considerata una gerarchia decrescente di quattro tipi di grammatiche:

1. *Grammatiche di tipo 0*: le regole di produzione sono arbitrarie.

2. *Grammatiche di tipo 1*: ogni regola di produzione $\alpha \rightarrow \beta$ della grammatica deve essere tale che $l(\beta) \geq l(\alpha)$; è permessa la regola $S \rightarrow \varepsilon$, se S è l'assioma, a patto che S non compaia nella parte destra di nessuna altra regola. La grammatica presentata in Esempio 3.2 è di tipo 1.

3. *Grammatiche di tipo 2*: ogni regola di produzione $\alpha \rightarrow \beta$ della grammatica è tale che α è un metasimbolo. La grammatica presentata in Esempio 3.1 è di tipo 2. Un altro esempio è la seguente grammatica G, che genera il linguaggio di Dyck formato dalle parentesizzazioni corrette:

$$G = < \{(,)\}, \{S\} , \{S \rightarrow SS, S \rightarrow (S) , S \rightarrow \varepsilon \}, S>$$

4. *Grammatiche di tipo 3*: ogni regola di produzione della grammatica è del tipo $A \rightarrow \sigma B$, $A \rightarrow \sigma$ o $A \rightarrow \varepsilon$, dove A, B sono arbitrari metasimboli e σ un arbitrario simbolo terminale. La seguente grammatica G è di tipo 3 e genera il linguaggio $\{a^{2n} | n>0\}$:

$$G = < \{a\}, \{S,A\} , \{S \rightarrow aA , A \rightarrow aS , A \rightarrow a \}, S>$$

Ogni grammatica G genera un linguaggio L(G). Una classificazione delle grammatiche porta ad una naturale classificazione dei linguaggi.

Definizione 4.1 un linguaggio L è detto di *tipo k* (k=0,1,2,3) se esiste una grammatica G di tipo k tale che L = L(G). Indichiamo con R_k la classe dei linguaggi di tipo k (k=0,1,2,3).

Chiaramente le grammatiche di tipo k sono un sottoinsieme proprio delle grammatiche di tipo j, se k<j. Questo implica la seguente relazione di inclusione per le classi R_k :

$$R_3 \subseteq R_2 \subseteq R_1 \subseteq R_0$$

E' possibile mostrare che tale inclusione è propria:

Proposizione 4.1 $R_3 \subset R_2 \subset R_1 \subset R_0$

Dimostrazione:

- $R_3 \subset R_2$: come mostreremo in seguito, il linguaggio $\{a^n b^n \mid n \geq 1\} \notin R_3$; per contro $\{a^n b^n \mid n \geq 1\} \in R_2$ in quanto generabile dalla grammatica $G = < \{a,b\}, \{S\} , \{S \rightarrow aSb , S \rightarrow ab\}, S>$ che è di tipo 2.
- $R_2 \subset R_1$: come mostreremo in seguito, il linguaggio $\{a^n b^n b^n \mid n \geq 1\} \notin R_2$; per contro tale linguaggio è generato da una grammatica di tipo 1 (si veda Esempio 3.2)

- $R_1 \subset R_0$: il linguaggio A proposto in Teorema 2.1 è ricorsivamente numerabile ma non è ricorsivo. Da Teorema 3.1, esiste una grammatica di tipo 0 che genera A e quindi $A\in R_0$. Se ora proviamo che ogni linguaggio $L\in R_1$ è ricorsivo, possiamo concludere che $A\notin R_1$.
  Per provare che ogni linguaggio $L\in R_1$ è ricorsivo, fissiamo un linguaggio L generato da una grammatica $G= < \Sigma,Q,P,S>$ di tipo 1. Data una parola $w\in\Sigma^*$, si consideri il grafo orientato finito GR(w) che ha come insieme di vertici le parole $z\in(\Sigma \cup Q)^*$ con $l(w)\geq l(z)$ e come archi le coppie (x,y) per cui $x \Rightarrow_G y$.
  Osserviamo che $w\in L$ se e solo se c'è una derivazione $S\Rightarrow_G w_1$, ..., $w_i\Rightarrow_G w_{i+1}$ .., $w_m \Rightarrow_G w$ con $l(w)\geq l(w_m)\geq\ldots\geq l(w_1)\geq l(S)=1$, poiché se $\alpha\rightarrow\beta$ è una regola di produzione della grammatica vale $l(\beta)\geq l(\alpha)$. Possiamo concludere che $w\in L$ se e solo se in GR(w) c'è un cammino da S a w.
  Un algoritmo riconoscitore per L è allora il seguente:

  Input ($w\in\Sigma^*$)
  1. Costruisci GR(w)
  2. If "in GR(w) c'è un cammino da S a w" then return(1) else return(0) □

I linguaggi di tipo 0 coincidono, come abbiamo visto, coi linguaggi ricorsivamente numerabili.

I linguaggi di tipo 1 sono anche detti *dipendenti dal contesto* o *contestuali*, mentre quelli di tipo 2 sono detti anche *liberi dal contesto* o *non contestuali* . Questa terminologia deriva dal fatto che una regola $A\rightarrow\beta$, con A metasimbolo e β parola non vuota, è detta *non contestuale*, mentre la regola $xAy\rightarrow x\beta y$, con $xy\in(\Sigma \cup Q)^+$, è detta *contestuale*. La prima può essere applicata alla parola gAh, producendo la parola $g\beta h$, per qualsivoglia g,h , mentre la seconda può essere applicata alla parola gAh, producendo anche in questo caso la parola $g\beta h$, ma solo in certi contesti, e precisamente se x è suffisso di g e y è prefisso di h.
Ogni grammatica con regole di produzione contestuali è detta *contestuale* e un linguaggio generato da una grammatica contestuale è detto *dipendente dal contesto*. Poiché ogni grammatica contestuale è chiaramente di tipo 1 ed è possibile provare che per ogni grammatica G di tipo 1 esiste una grammatica G' contestuale che genera lo stesso linguaggio, i linguaggi di tipo 1 sono esattamente i linguaggi dipendenti dal contesto.

I linguaggi di tipo 3, infine, sono detti anche *linguaggi regolari*.

5 Linguaggi regolari e liberi dal contesto

In questo corso l'interesse è dedicato ai linguaggi di tipo 2 o 3; introduciamo qui alcune nozioni e risultati specifici per queste classi.

Cominciamo col richiamare che un linguaggio è di tipo 3 se esiste una grammatica di tipo 3 che lo genera. E' tuttavia possibile che una grammatica G non di tipo 3 generi un linguaggio di tipo 3; in questo caso deve naturalmente esistere una grammatica G' di tipo 3 equivalente a G. Esistono quindi classi di grammatiche un po' più generali di quelle di tipo 3 che tuttavia generano linguaggi di tipo 3. Come esempio, consideriamo le grammatiche lineari a destra: una grammatica è detta *lineare a destra* se le sue produzioni sono del tipo $A\rightarrow xB$ o $A\rightarrow y$, con A,B metasimboli e x,y parole di simboli terminali (eventualmente ε). Chiaramente una grammatica di tipo 3 è lineare a destra, mentre in generale non è vero il viceversa. Vale tuttavia:

Proposizione 5.1 Se L è generato da una grammatica lineare a destra, allora L è di tipo 3.
Dimostrazione: Sia G una grammatica lineare a destra che genera L. Il metodo dimostrativo consiste nel trasformare la grammatica lineare G in nuove grammatiche equivalenti (che generano cioè lo stesso linguaggio L), fino ad ottenere una grammatica equivalente di tipo 3.
Data la grammatica lineare G:

1. per ogni regola del tipo $A \rightarrow \sigma_1 \ldots \sigma_m B$ $(m>1)$, si ottiene una grammatica equivalente eliminando tale regola dopo aver introdotto nuovi metasimboli $M_1, M_2, \ldots, M_{m-1}$ e nuove regole $A \rightarrow \sigma_1 M_1$, $M_1 \rightarrow \sigma_2 M_2, \ldots, M_{m-1} \rightarrow \sigma_m B$;
2. stesso discorso per ogni regola del tipo $A \rightarrow \sigma_1 \ldots \sigma_m$ $(m>1)$.

Si ottiene in tal modo una grammatica equivalente che contiene regole del tipo $A \rightarrow \sigma B$, $A \rightarrow \sigma$, $A \rightarrow \varepsilon$, $A \rightarrow B$. Questa grammatica non è di tipo 3 solo per la presenza di regole del tipo $A \rightarrow B$. Allo scopo di eliminare regole di questo tipo, si considerino tutte le derivazioni del tipo $F \Rightarrow_G^* H$, dove F e H sono metasimboli e si proceda:

a) per ogni regola $A \rightarrow \sigma B$, si considerano tutte le coppie di metasimboli F ed H per cui $F \Rightarrow_G^* A$ e $B \Rightarrow_G^* H$ e si aggiungono le regole $F \rightarrow \sigma H$; infatti risulta possibile nella grammatica la derivazione $F \Rightarrow_G^* A \Rightarrow_G \sigma B \Rightarrow_G^* \sigma H$, che equivale a riscrivere F come σH.
b) per ogni regola $A \rightarrow \sigma$ (o $A \rightarrow \varepsilon$) si aggiungono le regole $F \rightarrow \sigma$ (o $F \rightarrow \varepsilon$) per ogni metasimbolo F per cui $F \Rightarrow_G^* A$; infatti risulta possibile nella grammatica la derivazione $F \Rightarrow_G^* A \Rightarrow_G \sigma$ (o $F \Rightarrow_G^* A \Rightarrow_G \varepsilon$), che equivale a riscrivere F come σ (o ε).

Possiamo a questo punto eliminare tutte le regole del tipo $A \rightarrow B$, ottenendo una grammatica di tipo 3 equivalente a G.

Si osservi inoltre che ogni grammatica di tipo 3 può essere trasformata in una equivalente contenente solo regole del tipo $A \rightarrow \sigma B$, $A \rightarrow \varepsilon$. Basta introdurre un nuovo metasimbolo M e la regola $M \rightarrow \varepsilon$, sostituendo poi ogni regola del tipo $A \rightarrow \sigma$ con la regola $A \rightarrow \sigma M$. Vale dunque:

Proposizione 5.2 se L è di tipo 3, allora è generato da una grammatica di tipo 3 contenenti solo regole del tipo $A \rightarrow \sigma B$, $A \rightarrow \varepsilon$.

E' possibile trasformare ogni grammatica di tipo 3 in una equivalente contenente solo regole del tipo $A \rightarrow \sigma B$, $A \rightarrow \sigma$, eliminando le regole del tipo $A \rightarrow \varepsilon$? In generale no, perché se L è generato da una grammatica contenente solo regole del tipo $A \rightarrow \sigma B$, $A \rightarrow \sigma$, allora $\varepsilon \notin L$. Vale tuttavia:

Proposizione 5.3 se L è di tipo 3 con $\varepsilon \in L$, allora $L = L' \cup \{\varepsilon\}$, dove L' è generato da una grammatica con regole del tipo $A \rightarrow \sigma B$, $A \rightarrow \sigma$.
Dimostrazione: Sia G la grammatica di tipo 3 che genera L; per ogni regola in G di tipo $A \rightarrow \sigma B$ si aggiungano regole del tipo $A \rightarrow \sigma$ se $B \Rightarrow_G^* \varepsilon$, eliminando poi tutte le regole del tipo $A \rightarrow \varepsilon$. La grammatica ottenuta genera L'.

Una proposizione analoga, che diamo senza dimostrazione, vale per linguaggi di tipo 2:

Proposizione 5.4 se L è di tipo 2 con $\varepsilon \in L$, allora $L=L' \cup \{\varepsilon\}$, dove L' è generato da una grammatica con regole del tipo $A \rightarrow x$, con $x \in (\Sigma \cup Q)^+$.

Esempio 5.1

- Supponiamo che in una grammatica lineare ci sia la produzione $D \rightarrow bbaE$. Al fine di ottenere una grammatica equivalente avente ogni produzione nella forma $A \rightarrow \sigma B$ o $A \rightarrow \sigma$, con $\sigma \in \Sigma$, dobbiamo (punto 1. della proposizione 5.1) sostituire la produzione $D \rightarrow bbaE$ con le produzioni $D \rightarrow bC$, $C \rightarrow bF$, $F \rightarrow aE$, dove C, F sono due nuovi metasimboli.
- Considera la grammatica con assioma A e con le seguenti produzioni: $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, $A \rightarrow D$, $B \rightarrow aA$, $D \rightarrow a$. Per eliminare le produzioni del tipo $M \rightarrow N$, dobbiamo (punto a) della proposizione 5.1) considerare tutte le derivazioni del tipo $F \Rightarrow_G^* H$. In questo caso abbiamo che $X \Rightarrow_G^* Y$ e $Z \Rightarrow_G^* D$ per $X,Y,Z \in \{A,B,C\}$ e quindi possiamo eliminare le produzioni $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, $A \rightarrow D$ inserendo nella grammatica le produzioni $X \rightarrow aY$ per $X \in \{A,B,C\}$e $Y \in \{A,B,C,D\}$, e $Z \rightarrow a$ con $Z \in \{A,B,C,D\}$. Lasciamo al lettore il compito di semplificare la grammatica così ottenuta.