# Complessità

In informatica la complessità di calcolo (o computazionale) di un algoritmo è la quantità di risorse richieste per la sua esecuzione. La complessità computazionale di un problema è la minima complessità di un algoritmo (noto o ignoto) che risolve il problema. Le risorse di calcolo a cui siamo maggiormente interessati sono il tempo di calcolo (o semplicemente tempo) e la quantità di memoria utilizzata (spazio). Lo scopo dell'analisi della complessità di calcolo è studiare la complessità di problemi ed algoritmi con l'obiettivo principale di trovare algoritmi di soluzione sempre più efficienti.

Valutazione dell'efficienza degli algoritmi

Il tempo impiegato ci darebbe una misura della complessità dell'algoritmo; inoltre, avendo due algoritmi che risolvono lo stesso problema, potremmo stabilire qual è il più efficiente semplicemente eseguendoli i e confrontando i rispettivi tempi di esecuzione. Questo metodo di valutazione non è attendibile se non consideriamo le condizioni in cui si effettuano le prove. Bisogna tener conto:

- dell'elaboratore su cui il programma viene eseguito
- del linguaggio di programmazione e dal compilatore utilizzato
- dei dati di ingresso ai due programmi
- della significatività dei dati di ingresso (è necessario eseguire i due programmi più volte con dati differenti)

Concludiamo che utilizzando unità di tempo standard come i secondi non otteniamo una valutazione oggettiva del costo di esecuzione di un programma. Vogliamo che la complessità dell'algoritmo permetta di confrontare due algoritmi a livello di idea. L'analisi della complessità è anche uno strumento che ci consente di spiegare come si comporta un algoritmo man mano che l'input aumenta.

Calcolo del costo di esecuzione di un programma

Per ottenere una misura del costo che non dipenda dal particolare elaboratore utilizzato valutiamo il numero di operazioni elementari effettuate da una macchina ispirata all'architettura di von Neumann e completamente dedicata: in tal modo ci si libera dalla necessità di fissare caratteristiche hardware/software di riferimento.

Per la valutazione del costo di un algoritmo si assume che ogni operazione elementare abbia costo unitario. Il costo di una istruzione composta è pari alla somma dei tempi relativi alle sue singole componenti. Per le istruzioni di controllo di flusso e le istruzioni di ciclo i costi da imputare sono unitari per il test e tengono conto del numero di istruzioni elementari eseguite. In particolare

- il costo di esecuzione di ogni istruzione semplice (istruzione di assegnazione, di lettura, di scrittura) è pari al numero di celle di memoria a cui l'istruzione assegna un valore. Tutte le istruzioni semplici della nostra macchina hanno costo unitario;
- il costo di un'istruzione di ciclo è dato dalla somma del costo totale di esecuzione del test di fine ciclo e dal costo totale di esecuzione del corpo del ciclo;
- il costo di un'istruzione condizionale di tipo if è dato dal costo di esecuzione del test (calcolato con le medesime regole del caso precedente) più il costo di esecuzione della istruzione che viene eseguita se la condizione è vera;
- il costo di esecuzione di una attivazione di sottoprogramma (o procedura o funzione) è pari al costo di esecuzione di tutte le istruzioni che compongono il sottoprogramma.

Assumiamo il costo di due istruzioni uguale se è esatto a meno di un fattore costante. Un importante vantaggio di queste semplificazioni è quello di poter prescindere dal linguaggio di programmazione in cui viene scritto il programma e di poter far riferimento direttamente all'algoritmo descritto in linguaggio naturale.

Analisi nel caso peggiore in funzione delle dimensioni dell'input

Osserviamo che il costo di esecuzione di un programma dipende quasi sempre dai particolari dati di ingresso. Pertanto, per tener conto del numero di dati con cui si esegue il programma assumiamo che la dimensione dell'input rappresenti l'argomento della funzione che esprime il costo di esecuzione di un programma. Per dimensione dell'input si intende la quantità di memoria necessaria per memorizzarlo. Si noti che in molti casi il costo di esecuzione del programma dipende non solo dalle dimensioni dell'input, ma anche dai particolari dati di ingresso. Generalmente possiamo distinguere diversi casi: il caso migliore (quello meno costoso), il caso peggiore (quello più costoso), il caso medio. Il costo di un programma viene valutato in funzione delle dimensioni dell'input con riferimento al caso peggiore, cioè quello in cui l'esecuzione impiega più tempo e assumendo che il costo di ogni singola istruzione sia pari a uno.

Non sempre tutti i dati in input concorrono a determinare il costo di un algoritmo. Nel caso di una elevazione a potenza l'elemento che determinerà il costo è l'esponente. La base della potenza non sarà particolarmente rilevante.

Analisi asintotica e notazioni

L'efficienza degli algoritmi impiegati ha grande rilevanza nel caso di input di dimensioni grandi. Infatti, nei casi di problemi di "piccola taglia" ovvero problemi i cui input hanno dimensione ridotta, la potenza di calcolo disponibile rende trascurabili inefficienze algoritmiche. L'intrattabilità computazionale è da associare all'andamento asintotico (esponenziale) della funzione di costo piuttosto che a costanti moltiplicative o che a una possibile scarsa potenza di una piattaforma hardware/software. L'obiettivo primario dell'analisi del costo computazionale è quindi individuare l'andamento asintotico della funzione di costo. Consideriamo nel seguito alcune proprietà delle notazioni introdotte:

1. $f(n) \in O(g(n)) \Longleftrightarrow g(n) \in \Omega(f(n))$
2. $f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)) \Longleftrightarrow f(n) \in \Theta(g(n))$
3. $f(n) \in \Theta(g(n)) \Longleftrightarrow g(n) \in \Theta(f(n))$

Data una funzione g(n) definita su N, O(g(n)) rappresenta l'insieme delle funzioni che per valori maggiori di $n_0$ assumono un valore pari al più a $c \cdot g(n)$. Osserviamo che se $f(n) \in O(n)$ allora abbiamo che $f(n) \in O(n^2)$ e $\in O(n^3)$.

Data una funzione g(n) definita su N, Ω(g(n)) rappresenta l'insieme delle funzioni che per valori maggiori di $n_0$ assumono un valore almeno pari a $c \cdot g(n)$. Osserviamo che se $f(n) \in \Omega(n^2)$ allora abbiamo che $f(n) \in \Omega(n)$.

Concludiamo presentando alcune semplici regole che possono essere d'aiuto per confrontare due funzioni f e g:

1. Le costanti moltiplicative e additive non contano nella notazione O
2. Ai fini della notazione O ci interessa solo l'esponente più grande
3. Date due costanti a e b, $a \leq b$ allora abbiamo $n^a \in O(n^b)$
4. Un qualunque polinomio è sempre più piccolo di una funzione esponenziale
5. Una funzione logaritmica è sempre più piccola di una funzione polinomiale

Operazioni dominanti

Una proprietà particolarmente utile dell'istruzione dominante è quella di costituire l'istruzione più frequentemente eseguita dall'algoritmo. A tal proposito, visto che la definizione di istruzione dominante è applicabile a un'istruzione elementare, è bene precisare che, qualora nell'algoritmo che si sta analizzando ci siano chiamate ad altri algoritmi, diviene necessario immaginare che alla chiamata venga sostituito il corpo dell'algoritmo chiamato. In pratica, per individuare l'istruzione dominante basta riferirsi all'operazione più frequentemente eseguita dall'algoritmo (ad esempio guardando all'interno dei cicli più interni). La componente di costo associata all'istruzione dominante determinerà l'andamento asintotico del costo temporale dell'algoritmo.

Esempi analisi di complessità

Ricerca Sequenziale: Θ(z); Ricerca Binaria: Θ(log z); Potenza $\Theta(2^z)$; Test Primalità: $\Theta(2^{z/2})$; Ordinamento a bolle: $\Theta(n^2)$

Analisi di algoritmi ricorsivi

Valutare la complessità di un algoritmo ricorsivo è più complesso che nel caso degli algoritmi iterativi. Infatti, la natura ricorsiva della soluzione algoritmica dà luogo a una funzione di costo che è anch'essa ricorsiva. Questa equazione - detta equazione di ricorrenza - definisce una sequenza di valori a partire da uno o più valori iniziali dati; ogni ulteriore valore della funzione è definito come funzione dei valori precedenti. Esempio: $G(n) = 1$ se $n = 1$ e $G(n) = 2G(n - 1)$ se $n > 1$

Dato un algoritmo, in genere trovare la funzione di costo ricorsiva è piuttosto immediato. La parte meno semplice è la soluzione dell'equazione riformulandola in una di ricorrenza equivalente ma non ricorsiva.

Equazioni di ricorrenza

Iniziamo dalla ricerca binaria ricorsiva. Il corpo della funzione, chiamato su un vettore di dimensione n:

- esegue un test verificando se l'elemento cercato è presente nel vettore
- se tale test non è soddisfatto effettua una chiamata ricorsiva su un vettore di dimensioni n/2 elementi

Indicando con T (n) la complessità dell'algoritmo ricorsivo, possiamo esprimere la complessità dell'algoritmo mediante questa equazione di ricorrenza: $T(n) = T(n/2) + \Theta(1)$ e $T(1) = \Theta(1)$

La soluzione delle equazioni di ricorrenza può essere fatta con il metodo basato sullo srotolamento

$T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = 2 + T(n/4) = 2 + 1 + T(n/8) = 3 + T(n/8) = 3 + 1 + T(n/16) = 4 + T(n/16) \ldots$

$T(n) = k + T(n/2^k)$

Assumiamo che n sia potenza di 2. In questo caso continuiamo a srotolare la ricorsione fin quando $n/2^k = 1$; ora $n/2^k = 1$ implica $2^k = n$ e quindi $k = \log_2 n$. Se n non è potenza di due allora interrompiamo lo srotolamento non appena $n/2^k < 1$. Questo implica che $n/2^{k-1} < 2$ e quindi $k = \log_2 n + 1$. Abbiamo quindi mostrato che la soluzione dell'equazione nel caso della ricerca binaria è $T(n) = \log_2 n + 1 = O(\log_2 n)$.

Ordinamento per fusione (Mergesort)

Consideriamo ora l'algoritmo di ordinamento per fusione, detto Mergesort, un algoritmo ricorsivo che, nel caso peggiore, ha costo O(n logn) per ordinare una collezione di n elementi.

Dato un array di n elementi se n = 1 allora l'array è ovviamente ordinato e l'algoritmo termina; altrimenti, se n > 1, si eseguono i seguenti passi:

1. Ordina la prima metà dell'array
2. Ordina la seconda metà dell'array
3. Unisci (merge) le due parti ordinate in un unico array ordinato

In questo caso l'equazione di ricorrenza è la seguente: $T(n) = 1$ se $n = 1$ e $T(n) = n + 2T(n/2)$ se $n > 1$

Applicando il metodo dello srotolamento e assumendo che n sia potenza di due e che la fusione costi esattamente n:

$T(n) = n + 2T(n/2) = n + 2(n/2 + 2T(n/4)) = 2n + 4T(n/4) = 2n + 4(n/4 + 8T(n/8)) = 3n + 8T(n/8)$ …

$T(n) = kn + 2^k T(n/2^k)$

Il valore finale di k da considerare é quello per cui $n/2^k = 1$ e, quindi $k = \log_2 n$. Ponendo $k = \log_2 n$ nell'equazione precedente otteniamo che la soluzione dell'equazione di ricorrenza per il programma di ordinamento per fusione è pari a $(n \log_2 n + n)$. Osserviamo inoltre che Mergesort nel caso peggiore ha costo $\Omega(n \log n)$.

Calcolo del Massimo Comun Divisore, l'algoritmo di Euclide

Consideriamo il problema di decidere se due numeri interi a e b siano relativamente primi, ovvero se il loro massimo comun divisore (indicato con MCD (a, b)) sia uguale a 1. Un algoritmo con complessità temporale polinomiale per la risoluzione di tale problema, noto sotto il nome di algoritmo di Euclide, è il seguente (assume x>=y):

```
int mcd_algorithm (int x, int y) { if (y == 0) return x; else return mcd_algorithm (y, (x % y)); }
```

Per valutare la complessità temporale di tale algoritmo, mostriamo che sono sufficienti O(log b) chiamate ricorsive, avendo indicato con b il valore del parametro y alla prima chiamata. Osserviamo ora che i numeri sono interi e, quindi, il valore dimezza ogni due iterazioni, diventa un quarto ogni quattro iterazioni, diventa un sedicesimo ogni sei iterazioni e così via. In particolare, ogni $2^k$ iterazioni il valore di b diventa una frazione pari a $1/2^k$. Questo implica che il numero di attivazioni complessive è al massimo $2 \log b = O(\log b)$. Poiché ciascuna chiamata ricorsiva richiede tempo costante possiamo affermare che la complessità dell'algoritmo di Euclide è O(log b). Poiché per rappresentare i valori di ingresso b usiamo log b bit, possiamo concludere che l'algoritmo di Euclide ha complessità lineare nella dimensione dell'input.

Calcolo dei numeri di Fibonacci

Questo esempio è utile per illustrare come una diretta realizzazione di un algoritmo ricorsivo possa avere un costo (sia in termini di tempo che di memoria) non accettabile.

I numeri di Fibonacci sono così definiti: F (n) il valore della funzione di FIbonacci è definito dalla seguente equazione di ricorrenza: $F(n) = 1$ se $n < 3$ e $F(n) = F(n-1) + F(n-2)$ se $n \geq 3$

Una diretta implementazione della definizione porta al seguente programma ricorsivo:

```
def F(n) { if(N==0 || N==1) return  1; else{ F(n) = F(n-1)+ F(n-2) }}
```

Analizziamo ora il costo del programma precedente. Osserviamo che il costo di attivazione di F (n) è costante più il costo delle eventuali attivazioni ricorsive. Quindi in base ai ragionamenti fatti possiamo affermare che il costo di F (n), a meno di costanti moltiplicative e additive è pari a 1 più il numero di attivazioni ricorsive della funzione. Possiamo pertanto concludere che il costo dell'algoritmo è $O(2^n)$. Ricordiamo che la dimensioni dell'input è O(log n), pari al numero di bit sufficienti per memorizzare n e quindi concludiamo che il costo dell'algoritmo ricorsivo è esponenziale nel valore del parametro n e doppiamente esponenziale nelle dimensioni dell'input.

Consideriamo ora una versione iterativa per il calcolo dei numeri di Fibonacci. In questo caso è facile verificare che il costo è O(k) dove k è il numero di iterazioni del ciclo. È facile vedere che questo numero è pari a n. Quindi possiamo concludere che il costo del secondo programma è O(n), che ricordiamo è esponenziale nella dimensione dell'input.

Delimitazione superiore e inferiore alla complessità di problemi

Fra tutti gli upper bounds applicabili a un dato problema è naturalmente più interessante quello asintoticamente inferiore, mentre gli altri upper bounds vengono considerati banali. Di fatto, parlando dell'upper bound di un problema, si fa implicitamente riferimento al miglior algoritmo noto, quello con il costo asintotico minore. Quando accade che un problema ammette la stessa funzione f (n) come upper e lower bound diciamo che f (n) è la complessità intrinseca del problema e tutti gli algoritmi che lo risolvono con costo O (f(n)) vengono qualificati come ottimali.

Complessità inferiore dell'Ordinamento

Ci limitiamo a considerare algoritmi che utilizzano confronti fra dati per stabilire l'ordinamento come sono ad esempio gli algoritmi di ordinamento a bolle e di ordinamento per fusione precedentemente considerati. I possibili ordinamenti di n interi sono n! (possibili permutazioni di n elementi). Se utilizziamo algoritmi di ordinamento basati su decisioni, il numero minimo di test da effettuare deve permettere di distinguere la permutazione ordinata fra le n! possibili permutazioni. Dato che ciascuna decisione fornisce un singolo bit ne consegue che il numero di test da effettuare necessario sia $\log_2(n!)$. Dimostriamo poi che log2(n!) = Ω(n log2 n) e otteniamo che Ω(n log (n)) è un limite inferiore per la complessità temporale di qualsiasi algoritmo di ordinamento basato su confronti.

# Calcolabilità

Definizioni

Informalmente un insieme è una collezione di oggetti che sono gli elementi dell'insieme. Dato un insieme A denotiamo con P(A) l'insieme di tutti i sottoinsiemi di A che è detto insieme potenza. L'insieme P(A) include sia l'insieme vuoto che A fra i suoi elementi. Se un insieme finito A contiene n elementi allora P(A) contiene $2^n$ elementi.

Nel caso di insiemi infiniti il concetto di cardinalità è complesso e non facilmente intuibile. Per questa ragione lo studio di insiemi infiniti si basa essenzialmente sul confronto reciproco di insiemi. In particolare, dati due insiemi A e B, li confronteremo usando le proprietà di funzioni.

Dati due insiemi (finiti o infiniti) A e B e una funzione f: A → B diciamo che

- f è suriettiva se per ogni elemento b ∈ B esiste almeno un elemento di a ∈ A tale che f (a) = b;
- f è iniettiva dati a, a', a ∈ A, a' ∈ A, a != a' abbiamo f (a) != f (a')
- f è una corrispondenza biunivoca se f è suriettiva e iniettiva.

Dati due insiemi (non necessariamente finiti) A e B, abbiamo che

1. |A| ≥ |B| se esiste una funzione suriettiva f: A → B
2. |A| > |B| se esiste una funzione suriettiva f: A → B ma non esiste una funzione suriettiva g: B → A
3. |A| = |B| se esiste una corrispondenza biunivoca fra A e B.

Insiemi infiniti: insiemi numerabili e non numerabili

Un insieme infinito ben conosciuto è l'insieme dei numeri naturali che denotiamo con N e che contiene 0 e tutti gli interi positivi: N = {0, 1, 2, 3, . . .}. Nel seguito vedremo che non esiste un solo tipo di infinito ma è possibile definire infiniti gradi di infinito. Quello che faremo è confrontare insiemi utilizzando il metodo di prova introdotto da Cantor.

Insiemi numerabili

Un insieme A è numerabile se i suoi elementi possono essere ordinati: a1, a2, a3, a4 …, an, …

- N è un insieme infinito numerabile.
- Un insieme A è infinito numerabile se e solo esiste una corrispondenza biunivoca fra N e A.
- Un insieme è numerabile se e solo se è finito o infinito numerabile.

Le considerazioni precedenti ci permettono di mostrare che altri insiemi sono numerabili. L'insieme Z di tutti gli interi (positivi, negativi e 0) è numerabile perché possiamo enumerare gli interi positivi e negativi così: 0, 1, −1, 2, −2, 3, −3, …


Proviamo che l'insieme dei numeri razionali positivi $Q^+$ è numerabile. Ricordiamo la definizione di numero razionale: q è un numero razionale positivo se esistono due interi positivi n e m tale che q=n/m. Osserviamo che $|Q^+| \geq |N|$ mostrando una funzione suriettiva f: $Q^+ \rightarrow N$. Rimane quindi da dimostrare che $|N| \geq |Q^+|$. Dato che N e (N×N) sono entrambi numerabili è sufficiente mostrare una funzione suriettiva f: $(N \times N) \rightarrow Q^+$.

In questo caso possiamo concludere che la riga tratteggiata nella figura mostra una funzione suriettiva f: $(N \times N) \rightarrow Q^+$ e che quindi mostra che $|Q^+| \leq |(N \times N)|$. Dopo aver mostrato che i razionali positivi sono numerabili possiamo anche provare che i numeri razionali (positivi e negativi) sono numerabili nello stesso modo con cui abbiamo dimostrato che Z è numerabile. $|N| = |Z| = |N \times N| = |Q+| = |Q|$

Non uno ma tanti infiniti

In questa sezione vediamo il risultato fondamentale di Cantor che ha mostrato l'esistenza di insiemi infiniti non numerabili e quindi di cardinalità superiore ai naturali. L'insieme dei numeri reali R non è numerabile: $|R| > |N|$.

Prima di vedere la prova geniale di Cantor sulla non esistenza di una funzione suriettiva fra N e R notiamo che nel nostro caso dobbiamo mostrare che tutte le possibili funzioni f: N → R non sono suriettive: la prova di questo non è banale perché le possibili funzioni f: N → R sono infinite. Non potendo analizzare tutte le funzioni fra N e R la prova del teorema è per contraddizione. Assumiamo che esista una funzione suriettiva f fra i naturali e i reali. Ovviamente se i numeri reali sono numerabili lo sono anche i numeri reali in [0, 1) e, quindi, possiamo assumere che esiste una tabella T come la seguente che contiene in qualche ordine tutti i numeri reali fra 0 e 1.

Table 1: Ipotetico ordinamento dei numeri reali

| | | | | | | |
|---|---|---|---|---|---|---|
| $y_1$ | 0, | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | $a_{1,4}$ | $a_{1,5}$ | ... |
| $y_2$ | 0, | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | $a_{2,4}$ | $a_{2,5}$ | ... |
| $y_3$ | 0, | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ | $a_{3,4}$ | $a_{3,5}$ | ... |
| $y_4$ | 0, | $a_{4,1}$ | $a_{4,2}$ | $a_{4,3}$ | $a_{4,4}$ | $a_{4,5}$ | ... |
| $y_5$ | 0, | . | . | . | . | . | ... |
| . | 0, | . | . | . | . | . | ... |
| . | 0, | . | . | . | . | . | ... |

Ora mostriamo che esiste un numero reale che non fa parte della lista T, contraddicendo quindi l'ipotesi che T contenga tutti i numeri reali. Assumiamo che x = 0, $b_1b_2b_3b_4$ . . .; il valore di della i-esima cifra decimale $b_i$ di x è determinato sulla base di $a_{i,i}$, la i-esima cifra dell'i-esimo numero dell'ordinamento dei numeri reali. In particolare, se $a_{i,i}$ != 0 allora $b_i = a_{i,i} - 1$ se $a_{i,i} = 0$ allora $b_i = 1$. Abbiamo mostrato che x è diverso da ogni numero nella lista T. Quindi T non è una lista di tutti i numeri reali in [0, 1]. Possiamo concludere che l'insieme dei numeri reali ha cardinalità superiore a |N|.

Il metodo di prova proposto da Cantor è noto come diagonalizzazione: l'idea infatti utilizza la diagonale della tabella che rappresenta l'ipotetica lista dei numeri per definire un numero diverso da ogni numero nella lista. L'idea può essere applicata per mostrare altri risultati. Dato un insieme infinito A, $|P(A)| > |A|$.

Problemi indecidibili

In questa sezione confronteremo il numero di programmi con il numero dei possibili problemi e vedremo che l'insieme dei problemi diversi fra loro ha cardinalità maggiore del numero di programmi. Per questa ragione devono esistere necessariamente infiniti problemi per cui non abbiamo un programma di soluzione.

Il numero dei programmi è numerabile. Ovviamente il numero dei programmi che possiamo scrivere è infinito. Analizziamo quanti testi possiamo scrivere usando un computer o una macchina da scrivere, ogni testo è una sequenza di simboli della tastiera. Se includiamo tutti i possibili simboli otteniamo un insieme finito di simboli. Osserviamo ora che ogni programma è un testo e possiamo ordinare i testi con un ordinamento lessicografico. Non tutti i testi sono programmi ma provando che il numero di testi che possiamo scrivere ha cardinalità N abbiamo dimostrato anche che il numero di programmi ha cardinalità N (ricordando che se $A \subseteq B$ allora $|A| \leq |B|$).

Problemi decidibili e indecidibili

Un problema di decisione è una arbitraria domanda del tipo vero/falso relativo ad un insieme infinito di possibili ingressi. Per un dato problema l'insieme dei possibili input viene partizionato di un due insiemi, quello che soddisfa la proprietà richiesta e gli altri. I programmi che risolvono problemi decisionali prendono in input stringhe di caratteri e riconoscono se l'input verifica o meno una data proprietà: il programma fornisce in uscita il valore true se la stringa verifica la proprietà e false altrimenti.

Un problema di decisione T

- è decidibile se esiste un programma che termina sempre e riconosce esattamente le stringhe che verificano T
- è semidecidibile se esiste un programma che termina sempre quando la risposta è true. Non si richiede che il programma termini quando l'input non verifica la proprietà richiesta
- è altrimenti indecidibile

Mostriamo ora che il numero possibile di problemi di decisione ha cardinalità superiore a |N|. Per fare questo limitiamo la nostra attenzione ad una classe particolare di problemi: i problemi che chiedono di riconoscere insiemi dei numeri naturali. I programmi di questo tipo hanno come input un numero naturale y e il programma deve determinare se y verifica una data proprietà. Un esempio di problema di riconoscimento di insiemi è quello che chiede di riconoscere l'insieme S dei numeri pari.

I problemi di decisione sono un insieme non numerabile

Mostriamo ora che l'insieme dei problemi che chiedono di riconoscere insiemi dei numeri ha cardinalità superiore a N e quindi esistono insiemi indecidibili. Il numero dei problemi diversi fra loro ha cardinalità strettamente superiore a |N|.

Per dimostrare il teorema usiamo il metodo della diagonalizzazione. Assumiamo per contraddizione che per ogni insieme S dei numeri naturali esista un programma che decide S. Dato che sia i programmi che i possibili input sono numerabili possiamo costruire una matrice M, con infinite righe contenenti i programmi e infinite colonne contenenti possibili input, che rappresenta tutti i possibili risultati di tutti i programmi che riconoscono insiemi dei numeri naturali. Ogni riga sarà tipo "true true false true false …". Prendendo un nuovo programma che ha come risultati quelli sulla diagonale ma invertiti otteniamo un nuovo programma non presente nella lista. Quindi, poiché il numero dei programmi ha cardinalità inferiore al numero dei problemi abbiamo così mostrato che esistono infiniti problemi indecidibili per cui non esiste un programma di soluzione.

Teorema di Cantor

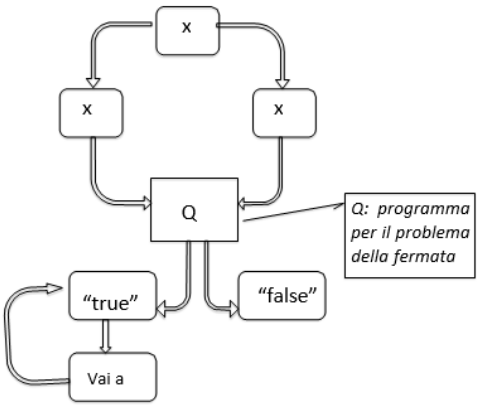
Sia L un insieme e $2^L$ l'insieme delle parti di L, allora non esiste una corrispondenza biunivoca tra i due. Dimostriamo dicendo per assurdo che esista f: L→$2^L$. Sia S={x appartenente a L | x non appartenente a f(x)} allora, per come abbiamo costruito S, per ogni x appartenente a L l'insieme S è diverso da f(x). Allora S è diverso da f(x) perché x appartiene a S se e solo se x non appartiene a f(x). Abbiamo trovato l'assurdo e dimostrato che l'insieme delle parti ha cardinalità maggiore di L. L'insieme dei linguaggi su un alfabeto ha cardinalità maggiore dell'insieme alfabeto* e quindi esistono linguaggi non decidibili.

Il problema della fermata

Esiste un programma Q che riceve in ingresso la specifica di un programma P e un possibile input y per P e decide se P si ferma con input y. Per una migliore comprensione della difficoltà del problema della fermata, osserviamo che è facile sapere se un programma P con input y si ferma: basta eseguire il programma e aspettare il tempo sufficiente alla sua esecuzione. La difficoltà del problema si verifica quando il programma non si ferma; in altre parole come possiamo decidere se un programma non si ferma?

Sia L un linguaggio di programmazione. Non esiste un programma Q scritto in L che, con input una coppia (x, y), dove x è la stringa che descrive un programma P scritto in L e y è una stringa di simboli di input per P, termina sempre in tempo finito e decide se P termina o no con input y.

Da un punto di vista matematico il problema della fermata chiede di scrivere un programma per calcolare la funzione




H(x, y) = true se il programma P termina con input y

H(x, y) = false altrimenti

Per dimostrare che il problema non è decidibile procediamo per contraddizione e assumiamo che Q sia una procedura che risolve il problema e che termina per ogni input (x, x). Se Q esiste allora esiste anche il programma il programma R la cui struttura è mostrata in figura. La modifica è evidenziata in basso a sinistra nella figura: nel caso Q dia in output true, il programma cicla. Sia quindi r la stringa di caratteri che descrive il programma R e analizziamo ora il comportamento di Q con input r. Per definizione Q con input (r, r) risolve il problema della terminazione di R con input r. Mostriamo ora che nessuna delle risposte true/false è possibile perché ciascuna porta ad una contraddizione.

In breve: chiamo Q un programma che risponde vero o falso se il programma in input x termina o no. Chiamo Qmod il programma Q con l'aggiunta di un ciclo infinito se Q restituisce vero. Qmod termina dicendo falso se x cicla all'infinito. Quindi Qmod termina solo se x non termina. Passando Qmod a Qmod, questo terminerà solo se Qmod non termina!

A questo punto la prova è completa: abbiamo assunto per contraddizione che Q risolve il problema della fermata e abbiamo mostrato che esiste un programma R per cui Q non fornisce una risposta corretta. Possiamo quindi concludere che cade l'assunzione iniziale che esista un programma Q che sia in grado di risolvere il problema della fermata. Pertanto, il problema della fermata è indecidibile.

Un esempio è la congettura di Goldbach, uno dei più vecchi problemi irrisolti nella teoria dei numeri. Essa afferma che ogni numero pari maggiore di due può essere scritto come somma di due numeri primi (che possono essere anche uguali). Il seguente programma Python si fermerà se e solo se la congettura di Goldbach è falsa.

def isprime(p): return all (p % i for i in range (2, p-1))

def Goldbach(n): return any ((isprime(p) and isprime(n-p)) for p in range (2, n-1))

n = 4; while (True): if not Goldbach(n): break; n+=2

## Macchine di Turing

Lo scopo di questo capitolo è investigare dispositivi di calcolo molto semplici e valutare le loro capacità di calcolo. Nel 1936 Alan Turing ha descritto un modello teorico di una macchina di calcolo. Sebbene molto semplice, la Macchina di Turing possiede notevoli proprietà. Una macchina di Turing non è meno potente di qualsiasi altro dispositivo di elaborazione oggi disponibile ed è in grado di risolvere qualunque problema risolubile con uno dei calcolatori che abbiamo a disposizione. La semplicità delle operazioni eseguibili con una macchina di Turing influenza il tempo di calcolo che risulta significativamente maggiore. Una macchina di Turing è definita da

1. un alfabeto finito di caratteri sul nastro Σ
2. un insieme finito S di stati
3. uno stato iniziale e due stati speciali: ACCETTA, RIFIUTA entrambi in S
4. una funzione di transizione δ: (Q − {ACCETTA, RIFIUTA}, Σ) → (Q, Σ, {D, S})

Esempio: Un contatore di parità.

δ(q0, 0) = (q0, 0, d), δ(q0, 1) = (q1, 1, d), δ(q0, □) = (ACCETTA, □, STOP )

δ(q1, 0) = (q1, 0, d), δ(q1, 1) = (q0, 1, d), δ(q1, □) = (RIFIUTA, □, STOP )

Una macchina di Turing continua l'esecuzione fino al raggiungimento di uno dei due stati ACCETTA/RIFIUTA; se non si raggiunge mai uno di questi stati il calcolo continua per sempre. Data una macchina di Turing possiamo quindi dividere i possibili input in tre categorie:

1. l'insieme degli input per cui la macchina termina nello stato ACCETTA
2. l'insieme degli input per cui la macchina termina nello stato RIGETTA
3. l'insieme degli input per cui la macchina non termina

Macchine di Turing e riconoscimento di linguaggi

Le macchine di Turing possono essere usate per riconoscere proprietà o per calcolare funzioni. Dato un alfabeto Σ, un linguaggio L è un sottoinsieme delle stringhe in Σ∗. Data una macchina di Turing M, il linguaggio accettato da M è L(M) = {w ∈ Σ∗|M con input w termina nello stato ACCETTA}. Un linguaggio L è Turing-accettabile se esiste una macchina di Turing che lo accetta, è Turing-decidibile se L è accettato da una Macchina di Turing che si ferma per ogni input.

Varianti delle macchine di Turing

Una macchina di Turing multinastro ha k nastri, ciascuno con la propria testa lettura/scrittura. In particolare, quando M è nello stato q legge i k simboli delle celle dove sono posizionati i diversi nastri e in funzione dello stato e dei k simboli letti, scrive su ciascun nastro, si sposta a sinistra o a destra su ogni nastro, ed entra in un nuovo stato. δ(q, a1, a2, ... , ak ) = (p, b1, ... , bk, M1, ... , Mk ) dove $b_i$ sono i nuovi simboli da scrivere sui nastri, $M_i$ sono le direzioni (destra o sinistra), in cui le testine si muovono, e p è lo stato successivo. Per ogni macchina di Turing con più nastri esiste una macchina di Turing con un solo nastro equivalente.

La macchina di Turing universale

Abbiamo visto che ogni macchina di Turing calcola una certa funzione e si comporta come un prefissato programma. La macchina di Turing universale (U), proposta da Turing nel 1936, è una macchina di Turing in grado di simulare una qualunque altra macchina di Turing arbitraria su input arbitrari. La macchina universale prende in input sia la descrizione della macchina M da simulare che un input I e calcola il risultato di M con input I. Per esemplificare come è realizzata U supponiamo che sia data una macchina di Turing M con un solo nastro e input I e vogliamo vedere come la macchina universale U realizzi i passi eseguiti da M con input I. La macchina universale U ha tre nastri:

| stato | simbolo | stato | simbolo | movimento | codifica |
|---|---|---|---|---|---|
| q0 | 0 | q0 | 1 | R | 0Z0UR |
| q0 | 1 | q0 | 0 | R | 0U0ZR |
| q0 | □ | q2 | □ | R | 0B10BR |
| q2 | 0 | q2 | 0 | L | 10Z10ZL |
| q2 | 1 | q2 | 1 | L | 10U10UL |
| q2 | □ | q1 | □ | R | 10B1BR |

Codifica: 0Z0UR0U0ZR0B10BR10Z10ZL10U10UL10B1BR

1. il primo nastro è di sola lettura e contiene la descrizione di M secondo una rappresentazione ad esempio come in figura
2. il secondo nastro è usato da U per la simulazione della macchina M (in altre parole rappresenta la memoria di lavoro) e contiene lo stato in cui si trova la macchina M
3. il terzo nastro rappresenta il nastro usato dalla macchina M e all'inizio contiene l'input I

Descriviamo ora la funzione di transizione di U. Non descriveremo U in dettaglio ma ci limiteremo a descrivere i passi essenziali con cui U esegue un passo di calcolo di M. La simulazione di un passo di calcolo di M richiede l'esecuzione di molti passi da parte di U che raggruppiamo nel seguente modo:

1. all'inizio U si posiziona sul primo carattere non vuoto del secondo nastro e legge la stringa di caratteri relativa allo stato q in cui si trova M e sul terzo nastro il carattere b letto dalla testina di M;
2. con queste due informazioni U ricerca nel primo nastro l'informazione relativa alla transizione dalla coppia (q, b) nella tabella di transizione di M; supponiamo che $\delta(q, b) = (qt, bt, M)$;
3. U completa l'esecuzione di un passo di calcolo di M scrivendo qt sul secondo nastro, scrivendo bt sul terzo nastro e spostando la testina del terzo nastro a destra o a sinistra a seconda del valore indicato;
4. U verifica se la simulazione di M è terminata; per fare questo verifica se lo stato qt attualmente memorizzato sul secondo nastro è uno stato finale (ACCETTA o RIGETTA). Se qt non `e uno stato finale allora U ritorna ad eseguire il passo 1, altrimenti termina.

La tesi di Church-Turing

Turing fu stimolato nelle sue ricerche dalle idee di David Hilbert e Kurt Gödel. Hilbert non era interessato alla verità di una singola affermazione e si chiedeva se "è possibile che ogni proposizione matematica possa essere provata vera o confutata". La seconda domanda che si poneva Hilbert era se fosse possibile identificare un metodo automatico per provare la verità o la falsità di una proposizione matematica. Gödel ha distrutto la prima speranza di Hilbert mostrando l'esistenza di proposizioni matematiche che erano indecidibili, il che significa che potrebbero essere né dimostrate né confutate. Chiaramente il risultato di Gödel ha reso necessario adattare e riformulare anche il secondo obiettivo che è stato riformulato in quello di stabilire la decidibilità, piuttosto che la verità, di una proposizione.

Modelli di calcolo equivalenti

Turing non era il solo ricercatore a cercare un modello di calcolo in grado di eseguire in modo automatico tutto quanto fosse decidibile. Quando Alonzo Church propose il suo modello congetturò che ogni funzione che potesse essere calcolata da un algoritmo può essere definita usando il λ-calcolo. Poi, in modo indipendente, Turing ha formulato la tesi che ogni algoritmo può essere programmato su una macchina di Turing. Poiché i modelli di calcolo proposti da Church e Turing sono equivalenti, allora le due tesi esprimono la stessa affermazione che oggi è nota come tesi di Church-Turing: Tutto quello che si può calcolare lo si può calcolare con le macchine di Turing. In particolare, un problema di decisione è decidibile solo se esiste una macchina di Turing che riconosce il linguaggio associato e una funzione è calcolabile solo se esiste una macchina di Turing che la calcola.

Macchine di Turing non deterministiche

Una macchina di Turing non deterministica è una macchina di Turing a cui non viene imposto il vincolo che, dato uno stato della macchina e un simbolo letto dalla testina, la transizione da eseguire sia univocamente determinata. Quindi, il grafo delle transizioni di una macchina di Turing non deterministica può includere un arco uscente dallo stato q la cui etichetta contiene almeno due triple $(\sigma, \tau1, m1)$ e $(\sigma, \tau2, m2)$ tali che $\tau1 \neq \tau2$ e $m1 \neq m2$. La computazione di una macchina di Turing T non è più rappresentabile mediante una sequenza di configurazioni. Tuttavia, essa può essere vista come un albero, detto albero delle computazioni, i cui nodi corrispondono alle configurazioni di T e i cui archi corrispondono alla produzione di una configurazione da parte di un'altra configurazione. Ciascuno dei cammini (finiti o infiniti) dell'albero che parte dalla sua radice (ovvero, dalla configurazione iniziale) è detto essere un cammino di computazione. L'insieme delle stringhe accettate da T è detto essere il linguaggio accettato da T ed è indicato con L(T).

Equivalenza tra macchine deterministiche e non deterministiche

Il prossimo teorema mostra che l'uso del non determinismo non aumenta il potere computazionale del modello di calcolo delle macchine di Turing: "Sia T una macchina di Turing non deterministica. Esiste una macchina di Turing deterministica T' tale che L(T) = L(T')".

Dimostrazione. Dobbiamo visitare l'albero in ampiezza o per livelli: la realizzazione di una tale visita può essere ottenuta utilizzando uno dei due nastri come se fosse una coda. In particolare, per ogni input x, T' esegue la visita in ampiezza dell'albero delle computazioni di T con input x, facendo uso dei due nastri nel modo seguente:

- Il primo nastro viene utilizzato come una coda in cui le configurazioni di T (codificate in modo opportuno) vengono inserite, man mano che sono generate, e da cui le configurazioni di T sono estratte per generarne di nuove: tale nastro viene inizializzato inserendo nella coda la configurazione iniziale di T con input x.
- Il secondo nastro viene utilizzato per memorizzare la configurazione di T appena estratta dalla testa della coda e per esaminare tale configurazione in modo da generare le configurazioni da essa prodotte.

Fintanto che la coda non è vuota, T' estrae la configurazione in testa alla coda e la copia sul secondo nastro: se tale configurazione è finale, allora T' termina nel suo unico stato finale. Altrimenti, ovvero se la configurazione estratta dalla coda non è finale, T' calcola le configurazioni che possono essere prodotte da tale configurazione e le inserisce in coda alla coda. Se la coda si svuota allora tutti i cammini di computazione sono stati esplorati e nessuno di essi è terminato in una configurazione finale: in tal caso, quindi, T' può terminare in uno stato non finale. Chiaramente, T' accetta la stringa x se e solo se esiste un cammino accettante all'interno dell'albero delle computazioni di T con input x: quindi, L(T) = L(T') e il teorema risulta essere dimostrato.

## Problemi trattabili e problemi intrattabili

La classe P

La classe P è l'insieme dei linguaggi L per i quali esiste un programma Python che decide L e per cui il tempo di esecuzione su input di dimensione n è $O(n^k)$ per qualche $k \geq 0$. Osserviamo che nella definizione precedente la restrizione relativa ad un programma Python non è restrittiva:

La classe P è l'insieme dei linguaggi L per i quali esiste una macchina di Turing con un solo nastro deterministica che decide L e per cui il numero di passi di calcolo su input di dimensione n è $O(n^k)$ per qualche $k \geq 0$ (polinomiale).

Le due definizioni precedenti sono equivalenti. Infatti, è possibile dimostrare che esiste una macchina di Turing che risolve un problema (o riconosce un linguaggio) in tempo polinomiale nella lunghezza dell'input se e solo se esiste un programma in un linguaggio di programmazione che risolve il problema e richiede tempo polinomiale. Se il nostro obiettivo è quello di usare il calcolatore per risolvere problemi la cui descrizione includa un numero relativamente alto di elementi, allora è necessario che l'algoritmo di risoluzione abbia una complessità temporale polinomiale: per questo motivo, la classe P è generalmente vista come l'insieme dei problemi risolvibili in modo efficiente. Sono esempi di linguaggi in P la ricerca di un elemento in un punto, il calcolo del valore di un polinomio in un punto, il calcolo del massimo comun divisore di due numeri sono esempi di programmi in P.

La classe NP

Viene introdotta anche una nuova classe di linguaggi, più estesa della classe P. La classe NP è l'insieme dei linguaggi per cui esiste una macchina di Turing nondeterministica che in tempo polinomiale decide se x appartiene o meno a L. Ovviamente tutti i problemi che appartengono alla classe P appartengono anche a NP. Infatti, una macchina di Turing deterministica è un caso speciale di macchina nondeterministica. Quindi possiamo dire che la classe P è inclusa nella classe NP: $P \subseteq NP$ Non sappiamo se l'equivalenza precedente sia stretta o meno. Il problema SAT è un esempio di problema in NP (non sappiamo se appartenga a P). Altri esempi NP:

- il problema di partizionare un insieme in due parti uguali. In particolare, sono dati n numeri interi a1, a2, ..., an e ci chiediamo se sia possibile dividerli in due sottoinsiemi disgiunti tale che la somma dei due sottoinsiemi sia uguale fra loro.
- il problema di colorare una mappa. Data una carta geografica che rappresenta nazioni, ci chiediamo se sia possibile colorare tutti gli stati della mappa con tre colori in modo tale che due stati confinanti abbiano colori diversi.
- il problema di trovare il percorso più breve per un insieme di punti. In questo caso sono dati n punti nel piano e un valore x e ci chiediamo se esiste un percorso che collega tutti i punti dati con segmenti rettilinei, abbia lunghezza minore di x e non passi due volte per lo stesso punto.

Definizione alternativa: La classe NP è l'insieme di tutti i linguaggi L per cui, se x appartiene a L, allora esiste un certificato c(x) tale che c(x) ha lunghezza polinomiale in |x| ed esiste una MdT deterministica che con input x e c(x) verifica che x appartiene a L in tempo polinomiale in |x| e |c(x)|.

Oltre NP

Introduciamo due classi di complessità che includono la classe NP e che molto probabilmente sono strettamente più grandi di quest'ultima. La prima classe è la classe EXP, che include tutti i problemi e i linguaggi decidibili in tempo esponenziale; la seconda classe è la classe PSPACE, che include tutti i linguaggi decidibili usando un numero polinomiale di celle di memoria. Abbiamo le seguenti relazioni:

- NP ⊆ PSPACE. La dimostrazione di questa relazione può essere dimostrata nel seguente modo. Un algoritmo non deterministico che ha tempo di esecuzione polinomiale può utilizzare al massimo uno spazio polinomiale (utilizza nel caso peggiore un numero di celle di memoria pari al numero di passi di calcolo).
- PSPACE ⊆ EXP. Una macchina di Turing con n stati e che utilizza s celle del nastro può trovarsi al massimo in un numero di configurazioni diverse pari a $O(n2^s)$ (infatti se la macchina usa s celle del nastro le configurazioni diverse del nastro sono $O(2^s)$). Questo implica che se la macchina impiegasse più di $O(n2^s)$ passi, allora avremmo che necessariamente esistono due configurazioni del nastro identiche in cui la macchina si trova nello stesso stato. Questo implica che necessariamente la macchina non termina e cicla per sempre.

Conclusioni

In conclusione, valutiamo i criteri in base ai quali conviene scegliere l'algoritmo di soluzione. Infatti, quando dobbiamo risolvere un problema possiamo generalmente scegliere tra diversi algoritmi di soluzione e non sempre è chiaro quale sia la scelta migliore. Le proprietà che desideriamo dal nostro algoritmo sono principalmente due:

1. l'algoritmo deve essere semplice, in modo tale da facilitarne la comprensione, la programmazione e la correzione del programma;
2. l'algoritmo deve avere una complessità il più bassa possibile.

La prima proprietà fa riferimento al costo umano necessario per la scrittura del programma, mentre la seconda fa riferimento al costo di esecuzione del programma. L'unico criterio che possiamo dare è il seguente:

- quando il programma deve essere eseguito poche volte su input aventi piccole dimensioni, allora è generalmente più importante cercare la semplicità del programma, perché il costo umano prevale sul costo della macchina;
- quando, invece, il programma deve essere eseguito molte volte su input aventi dimensioni grandi, allora conviene cercare di ottenere algoritmi efficienti perché il risparmio di risorse della macchina può convenire rispetto al maggior tempo che si richiede per la progettazione e la scrittura del programma.

Problema di Knapsack

L'input è una lista di oggetti ciascuno con un peso ed un valore benefico. L'obiettivo è scegliere gli elementi che danno il massimo valore benefico senza eccedere il massimo peso consentito. Un algoritmo di forza bruta potrebbe risolvere il problema controllando le $2^n$ possibili soluzioni. Problema di decisione associato: esiste un insieme di oggetti da inserire nello zaino di peso non superiore a W e beneficio almeno pari a B?

Problema del commesso viaggiatore

È data una mappa di una regione con le distanze fra le città e una città di partenza. Si vuole trovare il percorso più breve per visitarle tutte una sola volta e tornare a quella di partenza. Si possono utilizzare i grafi pesati. Un algoritmo di forza bruta controlla le n! possibili combinazioni.

Colorazione di grafi

Dato un grafo G dobbiamo colorare i suoi nodi in modo tale che nodi adiacenti abbiano colori diversi. Il numero cromatico C(G) è il numero minimo di colori necessari. Possiamo associare un problema di decisione: dato il grafo G e un intero k, posso colorare G con k colori? C(G) sarà compreso fra 1 e il numero di nodi di G.

Tipi di problemi

I problemi decisionali hanno come risposta SI o NO. I problemi di ottimizzazione hanno come risposta un valore. Ogni problema di ottimizzazione può essere riformulato come un problema di decisione. Applicando più volte il problema di decisione possiamo risolvere il problema di ottimizzazione. Viene introdotto il comando Choice(I) che sceglie nondeterministicamente un elemento dell'insieme I. Il numero di problemi decisionali da risolvere è polinomiale nelle dimensioni dell'input. Se un problema di decisione è facile (polinomiale) lo sarà anche quello di ottimizzazione, se è difficile lo sarà anche quello di ottimizzazione. Possiamo quindi limitarci a studiare la complessità dei problemi di decisione. Il certificato di un problema decisionale è un insieme di informazioni che permette di verificare se P ha soluzione. In molti casi il certificato rappresenta la soluzione del problema stesso. Nel caso di un problema SAT il certificato è una assegnazione di valori Vero/Falso alle variabili.

SAT

Il problema SAT consiste nel decidere se una data formula booleana è soddisfacibile. Non si conoscono algoritmi polinomiali che risolvono questo problema, ma possiamo osservare che se una formula F con n variabili e m clausole è soddisfacibile, allora esiste un'assegnazione che soddisfa F. Un'assegnazione altro non è che una stringa di n valori

binari, pertanto tale assegnazione può essere codificata con una stringa di lunghezza polinomiale nella lunghezza della codifica di F. Inoltre, esiste un semplice algoritmo polinomiale che verifica se un'assegnazione di valori di verità soddisfa la formula: tale algoritmo deve semplicemente sostituire alle variabili i valori assegnati e verificare che ogni clausola sia soddisfatta. Applicando questo algoritmo per ogni possibile combinazione avrà un costo esponenziale ($p(n)\ 2^n$).

P è uguale a NP?

Non abbiamo una dimostrazione che smentisca quest'affermazione ma pensiamo che la classe P sia inclusa propriamente nella classe NP. È come chiedersi se trovare una soluzione a un problema sia in generale più difficile o no che verificare se una soluzione è corretta.

Riduzioni

Un linguaggio L1 è riducibile a un linguaggio L2 se esiste una funzione f: {0,1}* → {0,1}* detta riduzione, tale che, per ogni stringa binaria x, x appartiene a L1 se e solo se f(x) appartiene a L2. In questo caso se L2 è decidibile allora lo è anche L1. Se L1 non è decidibile allora non lo è neanche L2. Di base L1 non è più difficile di L2 e L2 è almeno difficile quanto L1. Intuitivamente diciamo che L1 non è più difficile di L2, ovvero che se esiste una MdT che decide L2 allora ne esiste anche una che decide L1. La riduzione va dal problema che so indecidibile a quello che voglio dimostrare indecidibile.

Riduzioni polinomiali e NP-completezza

Un linguaggio L1 è polinomialmente riducibile a un linguaggio L2 se esiste una riduzione da L1 a L2 calcolabile in tempo polinomiale e tale che ogni stringa x appartiene a L1 se e solo se f(x) appartiene a L2. Se L2 è risolubile in tempo polinomiale lo sarà anche L1. Se L1 non lo è allora non lo sarà neanche L2.

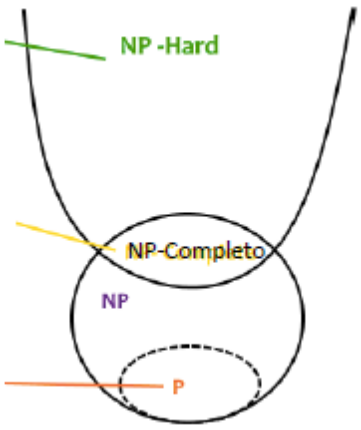
Un linguaggio L è NP-COMPLETO se appartiene a NP e se ogni altro linguaggio in NP è polinomialmente riducibile a L. Se dimostrassimo che un problema NP-completo è risolubile in tempo polinomiale allora avremmo dimostrato anche che P=NP. Il teorema SAT è NP-completo. Dimostrazione: dato un linguaggio L in NP, sappiamo che esiste un algoritmo con complessità temporale polinomiale V e un polinomio p tali che, per ogni stringa x, se è in L, allora esiste una stringa y di lunghezza non superiore a p(|x|) tale che V con x e y in ingresso termina in uno stato finale, mentre se x non appartiene a L, allora, per ogni sequenza y, V con x e y in ingresso termina in uno stato non finale. L'idea consiste nel costruire per ogni x, in tempo polinomiale, una formula booleana F(x,y) le cui uniche variabili libere sono p(|x|) variabili $y_n$ intendendo con ciò che la soddisfacibilità della formula dipende solo dai valori assegnati a tali variabili.

Se un linguaggio L non è necessariamente NP ma ogni altro linguaggio in NP è polinomialmente riducibile a L allora è NP-difficile.

Posso dimostrare che un linguaggio è NP-completo trovando una riduzione polinomiale da un altro linguaggio NP-completo (ad esempio SAT).

- Riduzione da SAT a 3SAT: data una formula F con k clausole si definisce una formula F' che è soddisfacibile se e solo se F è soddisfacibile (modifica clausole e aggiunge variabili).
    - Se k=1 mantengo solo la clausola l1 e tengo come problema D={{l1,y1,y2}, {l1,-y1,y2}, {l1,y1,-y2}, {l1,-y1,-y2}} questo è soddisfatto se e solo se l1 è soddisfatto.
    - Se k=2 D={{l1,l2,y}, {l1,l2,-y}} che è soddisfatto se e solo se l1 e l2 sono soddisfatti.
    - Se k>3 D={{l1,l2,y1}, {-y1,l3,y2}, {-y2,l4,y3}, {-y3,l5,l6}}.

  Una clausola è un insieme di variabili in OR, le clausole si scrivono {a,b,c}




- Riduzione da 3SAT a Programmazione intera: se possiamo tradurre 3SAT in PL e sappiamo risolvere facilmente PL allora sappiamo anche risolvere 3SAT (falso). Ad ogni variabile di 3SAT associamo 2 variabili v e w e le limitiamo a 0 o 1 (falso o vero). Imponiamo che la loro somma sia 1 (uno vero e uno falso). Per ogni clausola di 3SAT definiamo un vincolo associato. Abbiamo riformulato 3SAT come PI e quindi possiamo dire che PI è un problema NP-difficile. Non abbiamo dimostrato che sia NP e quindi non possiamo definirlo NP-completo (sappiamo però che lo è).
- Riduzione da 3SAT a 3-colorazione di grafi (è possibile colorare un grafo con 3 colori senza avere colori uguali adiacenti?). Costruiamo un grafo che simula un or e realizza una clausola. Alla fine dimostriamo che decidere se un grafo è colorabile con k>3 colori è NP-completo. Con k=1 o k=2 è un problema risolubile in tempo polinomiale. Per alcune classi è più semplice (ad esempio un albero può essere colorato solo con 2 colori).

# Linguaggi Automi Grammatiche

## Sintassi e semantica di un linguaggio

La sintassi specifica le regole secondo le quali una frase è corretta o meno nella lingua: una frase è sintatticamente corretta se è ottenuta applicando dei componenti che abbiano un senso strutturale. La sintassi non dice nulla circa il significato di una frase; questo è il compito della semantica. Possiamo facilmente trovare frasi contradditorie o frasi sintatticamente corrette che non hanno alcun significato.

L'analisi sintattica di un programma, che stabilisce se il programma rispetta le regole del linguaggio, è fatta automaticamente da un compilatore; se il programma non è corretto il compilatore fornisce gli errori sintattici. Gli errori semantici appaiono durante l'esecuzione del programma (a "run-time") quando il calcolatore interpreta il codice compilato.

Cominciamo elencando una serie di proprietà che sembrano evidenti nella definizione di un linguaggio:

- un numero di parole o simboli che sono utilizzati come elementi base (l'alfabeto)
- le regole che stabiliscono che solo certe sequenze di simboli di base sono frasi validi nella lingua

Possiamo dire che un linguaggio è un insieme di sequenze di simboli (le "frasi") che rispettano le regole del linguaggio.

## Alfabeti, stringhe, linguaggi

Un alfabeto è un insieme finito non vuoto di simboli (caratteri).

- Dato un alfabeto $\Sigma$, una sequenza finita di caratteri di $\Sigma$ è una stringa (o parola)
- Data una stringa x, $|x|$ rappresenta il numero di caratteri che la costituiscono
- L'operazione fondamentale sulle stringhe è la concatenazione
- La stringa vuota o nulla è la stringa di lunghezza zero
- L'insieme di tutte le stringhe definite sull'alfabeto $\Sigma$ (inclusa la stringa vuota) è denotato $\Sigma*$

Dato un qualunque alfabeto $\Sigma$ possiamo affermare che

- $\Sigma$ stesso è un linguaggio le cui stringhe sono gli elementi dell'alfabeto
- $\Sigma*$ è un linguaggio formato da tutte le possibili stringhe che si possono ottenere da $\Sigma$
- L'insieme che non contiene nessuna stringa, denotato $\emptyset$ è il linguaggio vuoto

## Approcci formali allo studio dei linguaggi

Lo studio formale dei linguaggi utilizza tre diversi approcci:

- un approccio algebrico, che mostra come costruire un linguaggio a partire da linguaggi più elementari utilizzando operazioni su linguaggi
- un approccio riconoscitivo, che definisce una "macchina" o un algoritmo di riconoscimento che ricevendo una stringa in input dice se essa appartiene o no al linguaggio
- un approccio generativo, che definisce attraverso una grammatica le regole strutturali che devono essere soddisfatte dalle stringhe che appartengono al linguaggio

Il nostro interesse per i linguaggi formali è anche motivato dalla necessità di scrivere compilatori e interpreti che hanno il compito di tradurre i programmi scritti in un linguaggio di programmazione ad alto livello in un linguaggio direttamente eseguibile da un calcolatore.

## Produzioni, simboli terminali e non terminali

Le grammatiche generative furono introdotte dal linguista Noam Chomsky negli anni Cinquanta. Risultarono estremamente utili nello sviluppo e nell'analisi di linguaggi di programmazione. Intuitivamente, una grammatica generativa specifica le regole attraverso le quali sia possibile, a partire da un simbolo iniziale, produrre tutte le stringhe appartenenti a un certo linguaggio.

Una grammatica generativa è una quadrupla (V, T, S, P) dove:

1. V è un insieme finito non vuoto di simboli non terminali
2. T è un insieme finito non vuoto di simboli terminali (V e T devono essere insiemi disgiunti)
3. S è un simbolo iniziale (anche detto simbolo di partenza oppure simbolo di frase) appartenente a V
4. P è un insieme finito di produzioni della forma $\alpha \rightarrow \beta$ dove $\alpha$ e $\beta$, dette forme sentenziali, sono sequenze di simboli terminali e non terminali con $\alpha$ contenente almeno un simbolo non terminale

Una produzione significa che ogni occorrenza della sequenza alla sinistra della produzione (ovvero α) può essere sostituita con la sequenza alla destra (ovvero β). Le frasi del linguaggio associato a una grammatica sono, dunque, generate partendo da S e applicando le produzioni fino a quando non restano solo simboli terminali. Si tratta di una definizione ricorsiva: esiste un caso iniziale (la sequenza composta dal solo simbolo iniziale S) e poi si applicano ripetutamente le possibili sostituzioni indicate dalle produzioni. L'insieme di tutte le sequenze di terminali $x \in T*$ che sono generabili da S formano il linguaggio generato dalla grammatica G, il quale è denotato con L(G).

Consideriamo la grammatica G = (V, T, S, P), dove V = {S}, T = {a, b} e P contiene le seguenti produzioni:

S → aSb; S → ab; È facile verificare che il linguaggio L(G) coincide con l'insieme delle stringhe $x = a^n b^n$, per n > 1.

La notazione BNF

La forma di Backus e Naur rappresenta in modo più compatto una grammatica. Mantiene i concetti di simboli non terminali, simboli terminali, assioma e produzioni. Si differenzia per una implicita rappresentazione che risulta più agevole per grammatiche complesse. Adotta la convenzione di rappresentare i simboli non terminali tra parentesi angolari "<" e ">". Al posto del simbolo → si utilizza il simbolo '::='. '|' che viene letto come 'oppure'. '+' denota la ripetizione da 1 ad un numero qualunque di volte del simbolo precedente

La gerarchia di Chomsky

In base alle restrizioni che vengono messe sul tipo di produzioni che una grammatica può contenere, si ottengono classi di grammatiche diverse. In particolare, Chomsky individuò quattro tipologie di grammatiche generative, definite nel modo seguente:

- Grammatiche regolari o di tipo 3. Ciascuna produzione è lineare a destra ovvero A → aB o A → a
- Grammatiche libere da contesto o di tipo 2. Ogni produzione deve essere del tipo A → α
- Grammatiche contestuali o di tipo 1. Ciascuna produzione deve essere del tipo α → β con |β| > |α|
- Grammatiche non limitate o di tipo 0. Nessun vincolo sussiste sulla tipologia delle produzioni

Si noti che una grammatica di tipo i, per i compreso tra 1 e 3, è anche una grammatica di tipo i – 1.

Linguaggi

Dato un linguaggio L diciamo che

1. L è decidibile se esiste una Macchina di Turing T che con input x si ferma in uno stato accettante se $x \in L$; se $x \notin L$ T si ferma e rifiuta x. In questo caso diciamo che T decide L.
2. L è semi-decidibile se esiste una Macchina di Turing T che con input x si ferma in uno stato accettante se $x \in L$. In questo caso diciamo che T semi-decide L. Nel caso in cui $x \notin L$ non si richiede che la macchina di Turing termini la computazione.

Se L e complemento di L sono semidecidibili, allora L è decidibile.

I linguaggi di tipo 0 sono semi-decidibili mentre i linguaggi di tipo 1 sono decidibili. Per dimostrare questi risultati introduciamo una variante delle macchine di Turing che ci consentirà di dimostrare più agevolmente che la classe dei linguaggi generati da grammatiche di tipo 0 coincide con quello dei linguaggi semi-decidibili.

Linguaggi di tipo 0 e semi-decidibilità

Per ogni linguaggio L di tipo 0, esiste una macchina di Turing non deterministica a due nastri che semi-decide L. Inoltre, se T è una macchina di Turing che semi-decide un linguaggio L, allora L è di tipo 0.

Dimostrazione. Sia G = (V, T, S, P) una grammatica non limitata che genera L. Una macchina di Turing T non deterministica a due nastri che accetta tutte e sole le stringhe di L può operare nel modo seguente.

1. Inizializza il secondo nastro con il simbolo S
2. Sia ϕ il contenuto del secondo nastro. Per ogni produzione α → β in P, non deterministicamente applica tale produzione a ϕ, ottenendo la stringa ψ direttamente generabile da ϕ
3. Se il contenuto del secondo nastro è uguale a x (che si trova sul primo nastro), termina nell'unico stato finale. Altrimenti torna al secondo passo.

Linguaggi di tipo 1

Se un linguaggio L è di tipo 1 allora esiste una macchina di Turing non deterministica T a tre nastri tale che L(T) = L e, per ogni input x, ogni cammino di computazione di T con input x termina.

Dimostrazione. Sia G = (V, T, S, P) una grammatica dipendente dal contesto che genera L. Una macchina di Turing T non deterministica a tre nastri che termina per ogni input e che accetta tutte e sole le stringhe di L. Opera in modo simile alla macchina non deterministica definita nella dimostrazione fatta sopra, ma ogni qualvolta cerca di applicare (in modo non deterministico) una produzione di G, verifica se il risultato dell'applicazione sia una stringa di lunghezza minore oppure uguale alla lunghezza della stringa x di input. Se così non è, allora T può terminare in uno stato non finale, in quanto siamo sicuri (in base alla proprietà delle grammatiche di tipo 1) che non sarà possibile da una stringa di lunghezza superiore a |x| generare x. Rimane anche da considerare la possibilità che la forma sentenziale generata da T rimanga sempre della stessa lunghezza a causa dell'applicazione ciclica della stessa sequenza di produzioni: anche in questo caso, dobbiamo fare in modo che T termini in uno stato non finale. A tale scopo, osserviamo che il numero di possibili forme sentenziali distinte di lunghezza k, è pari a y. La macchina T può evitare di entrare in un ciclo senza termine, mantenendo sul terzo nastro un contatore che viene inizializzato ogni qualvolta una forma sentenziale di lunghezza k viene generata sul secondo nastro per la prima volta. Per ogni produzione che viene applicata, se la lunghezza della forma sentenziale generata non aumenta, allora il contatore viene incrementato di 1: se il suo valore supera y, allora siamo sicuri che la computazione non avrà termine e, quindi, possiamo far terminare T in uno stato non finale. In conclusione, ogni cammino di computazione della macchina T termina e tutte e sole le stringhe generate da G sono accettate da almeno un cammino di computazione di T. Quindi, L(T) = L e il teorema risulta essere dimostrato.

Linguaggi regolari

Un compilatore è un programma che traduce un programma scritto in un linguaggio ad alto livello in uno scritto in linguaggio macchina. Nel seguito, il linguaggio ad alto livello che il compilatore riceve in ingresso è chiamato linguaggio sorgente, ed il programma in linguaggio sorgente che deve essere compilato è detto codice sorgente. Il particolare linguaggio macchina che viene generato è il linguaggio oggetto, e l'uscita del compilatore è il codice oggetto. Un compilatore è un programma molto complesso ed è realizzato mediante un numero separato di parti che lavorano insieme: queste parti sono note come fasi e sono cinque.

- Analisi lessicale: il compilatore scompone il codice sorgente in unità significanti dette token
- Analisi sintattica: il compilatore determina la struttura del programma e delle singole istruzioni
- Generazione del codice intermedio: il compilatore crea una rappresentazione interna del programma che riflette l'informazione non coperta dall'analizzatore sintattico
- Ottimizzazione: il compilatore identifica e rimuove operazioni ridondanti dal codice intermedio
- Generazione del codice oggetto: il compilatore traduce il codice intermedio nel codice oggetto

Analisi lessicale di linguaggi di programmazione

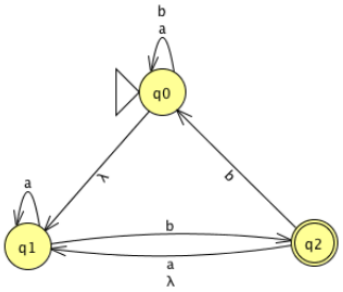
Il compito dell'analizzatore lessicale consiste nello scandire la sequenza del codice sorgente e scomporla in parti significanti, ovvero nei token di cui abbiamo parlato in precedenza. Il tipo generico, passato all'analizzatore sintattico, è detto token e le specifiche istanze del tipo generico sono dette lessemi. L'analizzatore lessicale deve isolare i token, prendere nota dei particolari lessemi e, quando un identificatore viene trovato, deve dialogare con il gestore della tabella dei simboli.

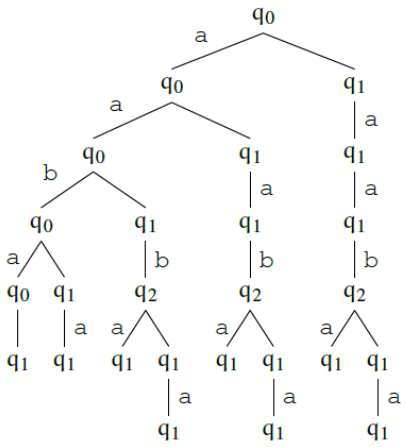
Automi a stati finiti

Chi sviluppa l'analizzatore lessicale definisce i token del linguaggio mediante espressioni regolari. Queste costituiscono una notazione compatta che indica quali caratteri possono far parte dei lessemi appartenenti a un particolare token e come devono essere messi in sequenza. L'analizzatore lessicale, tuttavia, viene meglio realizzato come un automa a stati finiti. Abbiamo già introdotto tali automi nella prima parte delle dispense: si tratta di macchine di Turing che possono solo leggere e spostarsi a destra e che al momento in cui incontrano il primo simbolo devono terminare accettando o rigettando la stringa di input. Infine, nel seguito indicheremo con L(T) il linguaggio accettato dall'automa a stati finiti T, ovvero l'insieme delle stringhe x per cui esiste un cammino nel grafo, che parte dallo stato iniziale e finisce in uno stato finale, le etichette dei cui archi formano x. Due automi a stati finiti T1 e T2 sono detti essere equivalenti se L(T1) = L(T2).

Automi a stati finiti non deterministici

Analogamente a quanto abbiamo fatto con le macchine di Turing nel precedente capitolo, possiamo modificare la definizione di automa a stati finiti consentendo un grado di non prevedibilità delle sue transizioni. Più precisamente, la transizione di un automa a stati finiti non deterministico può portare l'automa stesso in più stati diversi tra di loro. Come già osservato con le macchine di Turing non deterministiche, la computazione di un automa a stati finiti non deterministico corrisponde a un albero di possibili cammini di computazione: una stringa x è accettata dall'automa se tale albero include almeno un cammino che, dallo stato iniziale, conduce a uno stato finale e le etichette dei cui archi formino x. Nel caso degli automi a stati finiti, tuttavia, il non determinismo è generalmente esteso includendo la possibilità di λ-transizioni, ovvero transizioni che avvengono senza leggere alcun simbolo di input.

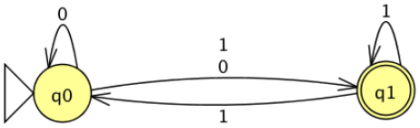


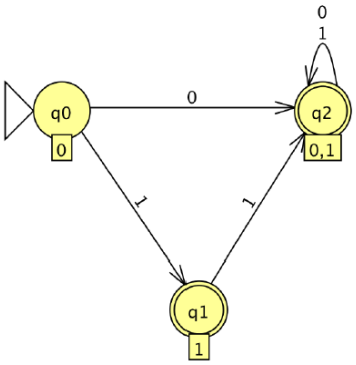



Automi deterministici e non deterministici

Sia T un automa a stati finiti non deterministico senza λ-transizioni → esiste un automa a stati finiti T' equivalente a T.

Dimostrazione. La dimostrazione procede definendo uno dopo l'altro gli stati di T' sulla base degli stati già definiti e delle transizioni di T (l'alfabeto di lavoro di T' sarà uguale a quello di T): in particolare, ogni stato di T' denoterà un sottoinsieme degli stati di T. Lo stato iniziale di T' è lo stato {q0} dove q0 è lo stato iniziale di T. La costruzione delle transizioni e degli altri stati di T' procede nel modo seguente. Sia Q = {s1, ..., sk} uno stato di T' per cui non è ancora stata definita la transizione corrispondente a un simbolo σ dell'alfabeto di lavoro di T e, per ogni i con i = 1, ..., k, sia $N(s_i, \sigma)$ l'insieme degli stati raggiungibili da $s_i$ leggendo il simbolo σ (osserviamo che S potrebbe anche essere l'insieme vuoto). Definiamo allora S = Unione N(si, σ) e introduciamo la transizione di T' dallo stato Q allo stato S leggendo il simbolo σ. Inoltre, aggiungiamo S all'insieme degli stati di T', nel caso non ne facesse già parte. Al termine di questo procedimento, identifichiamo gli stati finali di T' come quegli stati corrispondenti a sottoinsiemi contenenti almeno uno stato finale di T. È facile dimostrare che una stringa x è accettata da T se e solo se è accettata anche da T', ovvero L(T) = L(T').




Espressioni regolari

L'insieme delle espressioni regolari su di un alfabeto Σ è definito induttivamente come segue.

- Ogni carattere in Σ è un'espressione regolare.
- λ è un'espressione regolare.
- Se R ed S sono due espressioni regolari, allora
- La concatenazione R · S (o, semplicemente, RS) è un'espressione regolare.
- La selezione R + S è un'espressione regolare.
- La chiusura di Kleene R∗ è un'espressione regolare.
- Solo le espressioni formate da queste regole sono regolari.

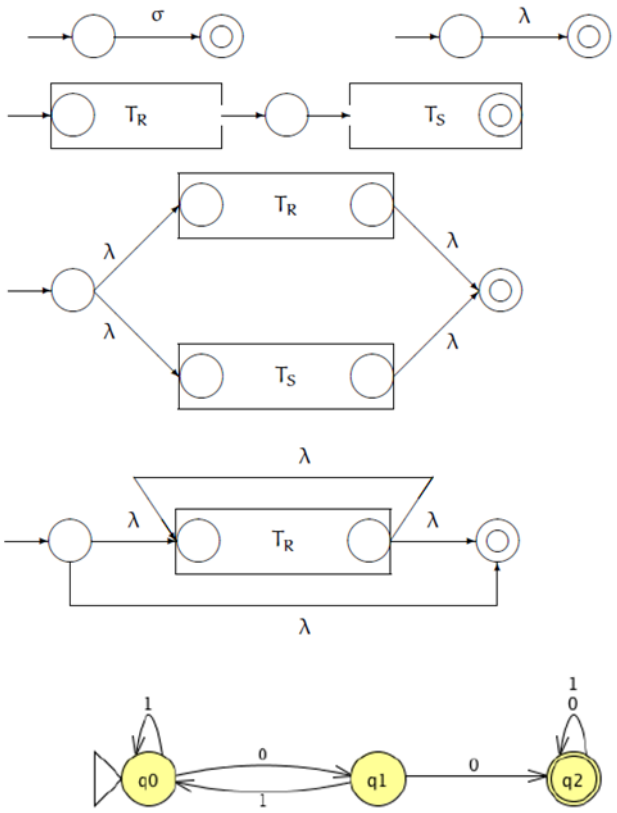
Nel valutare un'espressione regolare supporremo nel seguito che la chiusura di Kleene abbia priorità maggiore mentre la selezione abbia priorità minore: le parentesi verranno anche usate per annullare le priorità nel modo usuale. Un'espressione regolare R genera un linguaggio L(R). Osserviamo che la chiusura di Kleene consente zero concatenazioni. Due espressioni regolari R e S sono equivalenti se L(R) = L(S).

Le espressioni regolari sono in grado di generare solo un insieme limitato di linguaggi ma sono abbastanza potenti da poter essere usate per definire i token. Abbiamo già visto, infatti, che un token può essere visto come un linguaggio che include i suoi lessemi: rappresentando il token mediante un'espressione regolare, definiamo esattamente in che modo i lessemi sono riconosciuti come elementi del linguaggio.

Espressioni regolari ed automi a stati finiti

Siamo ora in grado di descrivere come può essere realizzato il secondo passo della catena che porta dalla definizione del token al corrispondente automa a stati finiti non deterministico: "Per ogni espressione regolare R esiste un automa a stati finiti non deterministico T tale che L(R) = L(T)".

Dimostrazione. La costruzione si basa sulla definizione induttiva delle espressioni regolari fornendo una macchina oppure un'interconnessione di macchine corrispondente a ogni passo della definizione. La costruzione, che assicura che l'automa ottenuto avrà un solo stato finale diverso dallo stato iniziale e senza transizioni in uscita, è la seguente:




- La macchina mostrata in alto a sinistra accetta il carattere σ ∈ Σ mentre quella mostrata in alto a destra accetta λ
- Date due espressioni regolari R e S, la macchina mostrata nella seconda riga accetta L(RS) dove $T_R$ e $T_S$ denotano due macchine che decidono L(R) ed L(S) e lo stato finale di $T_R$ è stato fuso con lo stato iniziale di $T_S$ in un unico stato
- Date due espressioni regolari R e S, la macchina nella terza riga accetta L(R+S) dove $T_R$ e $T_S$ denotano due macchine che riconoscono L(R) ed L(S), un nuovo stato iniziale è creato, due λ-transizioni da questo nuovo stato agli stati iniziali di $T_R$ ed $T_S$ sono aggiunte, un nuovo stato finale è stato creato e due λ-transizioni dagli stati finali di $T_R$ ed $T_S$ a questo nuovo stato sono aggiunte
- Data un'espressione regolare R, la macchina mostrata nella quarta riga della figura accetta L(R∗) dove $T_R$ denota una macchina che riconosce L(R), un nuovo stato iniziale e un nuovo stato finale sono stati creati, due λ-transizioni dal nuovo stato iniziale al nuovo stato finale e allo stato iniziale di $T_R$ sono state aggiunte e due λ-transizioni dallo stato finale di $T_R$ allo stato iniziale di $T_R$ e al nuovo stato finale sono state aggiunte

$$Q_0 \to 1Q_0 \quad Q_0 \to 0Q_1$$
$$Q_1 \to 1Q_0 \quad Q_1 \to 0Q_2 \quad Q_1 \to 0$$
$$Q_2 \to 0Q_2 \quad Q_2 \to 0 \quad Q_2 \to 1Q_2 \quad Q_2 \to 1$$

Automi a stati finiti e grammatiche regolari

Le grammatiche regolari o di tipo tre sono grammatiche le cui regole di produzione sono lineari a destra, ovvero del tipo X → a oppure del tipo X → aY. Il prossimo risultato mostra come tali grammatiche siano equivalenti agli automi a stati finiti e, quindi, alle espressioni regolari.

Dimostrazione. Sia T un automa a stati finiti e sia L = L(T) il linguaggio deciso da T. Costruiamo ora una grammatica regolare G che genera tutte e sole le stringhe di L. Tale grammatica avrà un simbolo non terminale per ogni stato di T: il simbolo iniziale sarà quello corrispondente allo stato iniziale. Per ogni transizione che dallo stato q fa passare l'automa nello stato p leggendo il simbolo a, la grammatica include la regola di produzione Q → aP, dove Q e P sono i due simboli non terminali corrispondenti agli stati q e p. Inoltre, se p è stato finale, allora la grammatica include anche la transizione Q → a. È facile verificare che L(T) = L(G). Viceversa, supponiamo che G sia una grammatica regolare e che L = L(G) sia il linguaggio generato da G. Definiamo un automa a stati finiti non deterministico T tale che L(T) = L: questo dimostra che esiste un automa a stati finiti equivalente a G. L'automa T ha uno stato per ogni simbolo non terminale di G: lo stato iniziale sarà quello corrispondente al simbolo iniziale di G. Per ogni produzione tipo X → aY di G, T avrà una transizione dallo stato corrispondente a X allo stato corrispondente a Y leggendo il simbolo a. Inoltre, per ogni produzione del tipo X → a di G, T avrà una transizione dallo stato X all'unico stato finale F leggendo il simbolo a. L(T) = L(G).

Da grammatiche di tipo 3 a espressioni regolari

Per ogni grammatica regolare G esiste una espressione regolare che definisce lo stesso linguaggio generato da G. Per costruirla si procede in due passi:

1. Si trasforma la grammatica in un sistema di equazioni contenenti simboli terminali e variabili. Ogni equazione sarà del tipo <variabile>=<espressione regolare estesa>. A →aB|c diventa A=aB+c, S→aS (ricorsiva) diventa S=a*, B→bB|c diventa B=b*+c
2. Si risolvono le equazioni sostituendo ai non terminali i terminali
3. Semplificazione della formula

**Esempio.** Sia data la grammatica con assioma A che genera il linguaggio delle stringhe che contengono un numero pari, anche 0, di a:

A → bA | aB | b  B → aA | bB | a

Otteniamo il sistema

A = bA + aB + b  B = bB + aA + a

- Si elimina la ricursione nella seconda equaz. B = b*(aA + a)
- si sostituisce nella prima A = bA + ab*(aA + a) + b
- si semplifica A = bA + ab*aA + ab*a + b
- si fattorizza A = (ab*a+b)A + ab*a + b
- e si termina eliminando di nuovo la ricursione

  A = (ab*a+b)*(ab*a + b) = (ab*a+b)$^+$

Linguaggi non regolari

Dimostriamo che un linguaggio non è regolare:

- Tutti i linguaggi finiti sono regolari
- Se un linguaggio infinito L è deciso da un automa a stati finiti T, questo automa deve necessariamente avere un numero finito di stati
- Un automa con numero finito di stati non è in grado di contenere infinita memoria

Il linguaggio L = {$a_n b_n$, n>0} non è regolare: per riconoscere la stringa l'automa deve essere in grado di contare quante "a" ci sono e verificare che ci siano tante "b" con un numero infinito di stati.

| Tipo di linguaggio | Tipo di produzioni | Modello di calcolo |
|---|---|---|
| Tipo 0 | $\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$ e $\beta \in (V \cup T)^*$ | Macchina di Turing |
| Contestuale | $\alpha \to \beta$ con $\alpha \in (V \cup T)^* V (V \cup T)^*$, $\beta \in (V \cup T)(V \cup T)^*$ e $\lvert\beta\rvert \geqslant \lvert\alpha\rvert$ | Macchina di Turing lineare |
| Libero da contesto | $A \to \beta$ con $A \in V$ e $\beta \in (V \cup T)(V \cup T)^*$ | |
| Regolare | $A \to aB$ e $A \to a$ con $A, B \in V$ e $a \in T$ | Automa a stati finiti |

Grammatiche di tipo 2: linguaggio libero dal contesto

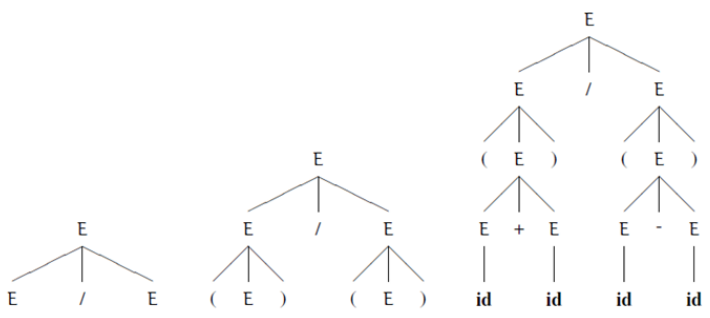
Sono adatte a descrivere la sintassi dei linguaggi di programmazione, sono più semplici del tipo 1 e ci permettono di costruire parser più efficienti. Sono più espressive del tipo 3 e consentono di descrivere la sintassi dei linguaggi di programmazione. I linguaggi liberi dal contesto sono riconosciuti da automi a stati finiti con l'ausilio di una pila. I linguaggi di programmazione appartengono a questa categoria ma con ulteriori restrizioni.

Analisi sintattica top-down

Consideriamo le espressioni aritmetiche formate facendo uso delle sole operazioni di somma, sottrazione, moltiplicazione e divisione. Una semplice grammatica libera da contesto per tali espressioni è la seguente:

E → E + E; E → E – E; E → E ∗ E; E → E/E; E → (E); E → id

Facendo uso di tale grammatica, analizziamo l'espressione (a + b)/(a − b).




Data una grammatica G = (V, N, S, P), un albero di derivazione è un albero in cui la radice è etichettata con S (simbolo iniziale), le foglie sono etichettate con simboli in V (simboli terminali) e per ogni nodo con etichetta un simbolo non terminale A, i figli di tale nodo sono etichettati con i simboli della parte destra di una produzione in P la cui parte sinistra sia A. Un albero di derivazione è detto di una stringa x se i simboli che etichettano le foglie dell'albero, letti da sinistra verso destra, formano la stringa x.

Grammatiche ambigue

Le grammatiche generative possono essere ambigue in quanto la stessa stringa può essere prodotta in diversi modi. Consideriamo l'espressione id + id ∗ id. Possiamo generarla usando la produzione E → E ∗ E, quindi applicando la produzione E → E + E e, infine, applicando ripetutamente la produzione E → id. Ma avremmo anche potuto iniziare applicando E → E + E, quindi E → E ∗ E e, infine, E → id. Questi due alberi sono chiaramente diversi: qual è quello giusto? Non possiamo rispondere a questa domanda se conosciamo solo la grammatica, in quanto entrambi gli alberi sono stati costruiti usando le sue produzioni e non vi è ragione perché entrambi non possano essere accettabili. Una grammatica nella quale è possibile analizzare anche una sola sequenza in due o più modi diversi è detta ambigua. La soluzione è quella di riscrivere la grammatica in modo da eliminare le ambiguità. Per esempio possiamo definire la seguente grammatica alternativa: E → E + T; E → E – T; E → T; T → T ∗ F; T → T/F; T → F; F → (E); F → id

Derivazioni destre e sinistre

Scegliendo il non terminale più a destra come quello da sostituire otteniamo una derivazione destra. Alternativamente, avremmo potuto selezionare il non terminale più a sinistra a ogni passo e avremmo ottenuto una derivazione sinistra. Notiamo che mentre è possibile costruire diverse derivazioni corrispondenti allo stesso albero di derivazione, le derivazioni sinistre e destre sono uniche. Ogni forma sentenziale che occorre in una derivazione sinistra è detta forma sentenziale sinistra e ogni forma sentenziale che occorre in una derivazione destra è detta forma sentenziale destra.

Parser top-down

Un parser top-down parte dalla radice dell'albero di derivazione e cerca di ricostruire la crescita dell'albero che porta alla data sequenza di simboli terminali: nel fare ciò, ricostruisce una derivazione sinistra. Il parser top-down deve iniziare dalla radice dell'albero e determinare in base alla sequenza di simboli terminali come far crescere l'albero di derivazione: inoltre, deve fare questo in base solo alla conoscenza delle produzioni nella grammatica e dei simboli terminali in arrivo da sinistra verso destra. L'albero viene ricostruito in pre-ordine.

Problemi:

Produzioni della forma $A \rightarrow A\alpha$ sono produzioni ricorsive a sinistra e nessun parser top-down è in grado di gestirle. Infatti, osserviamo che il parser procede "consumando" i simboli terminali: ognuno di tali simboli guida il parser nella sua scelta di azioni. Quando il simbolo terminale è usato, un nuovo simbolo terminale diviene disponibile e ciò porta il parser a fare una diversa mossa. I simboli terminali vengono consumati quando si accordano con i terminali nelle produzioni: nel caso di una produzione ricorsiva a sinistra, l'uso ripetuto di tale produzione non usa simboli terminali per cui, a ogni mossa nuova, il parser ha di fronte lo stesso simbolo terminale e quindi farà la stessa mossa. Questo problema affligge tutti i parser top-down comunque essi siano implementati. La soluzione consiste nel riscrivere la grammatica in modo che siano eliminate le ricorsioni sinistre. A tale scopo distinguiamo tra due tipi di tali ricorsioni: le ricorsioni sinistre immediate generate da produzioni del tipo $A \rightarrow A\alpha$ e quelle non immediate generate da produzioni del tipo $A \rightarrow B\alpha$ e $B \rightarrow A\beta$.

Per rimuovere le ricorsioni sinistre immediate, procediamo nel modo seguente (nel seguito assumiamo che la grammatica originale non contenga λ-produzioni, ovvero produzioni la cui parte destra sia uguale a λ). Per ogni non terminale A nella grammatica che presenta almeno una produzione ricorsiva sinistra immediata, eseguiamo le seguenti operazioni.

1. Separiamo le produzioni ricorsive a sinistra immediate dalle altre. Supponiamo che le produzioni siano le seguenti: $A \rightarrow A\alpha 1 \mid A\alpha 2 \mid A\alpha 3 \mid \cdots$ e $A \rightarrow \delta 1 \mid \delta 2 \mid \delta 3 \mid \cdots$
2. Introduciamo un nuovo non terminale A'.
3. Sostituiamo ogni produzione non ricorsiva $A \rightarrow \delta i$ con la produzione $A \rightarrow \delta iA'$.
4. Sostituiamo ogni produzione ricorsiva immediata $A \rightarrow A\alpha i$ con la produzione $A' \rightarrow \alpha iA'$.
5. Aggiungiamo la produzione $A' \rightarrow \lambda$.

Osserviamo che la rimozione di ricorsioni immediate introduce produzioni del tipo $A \rightarrow \lambda$. L'idea dell'algoritmo appena descritto è quella di anticipare l'uso delle produzioni non ricorsive che in ogni caso dovranno essere prima o poi utilizzate: così facendo, le ricorsioni sinistre immediate vengono trasformate, mediante l'uso di un nuovo simbolo non terminale, in ricorsioni destre immediate, le quali non creano problemi a un parser di tipo top-down.

Per rimuovere tutte le ricorsioni sinistre in una grammatica procediamo invece nel modo seguente:

1. Creiamo una lista ordinata A1, ..., Am di tutti i simboli non terminali.
2. Per j = 1, ..., m, eseguiamo le seguenti operazioni:
   a) per h = 1, ..., j − 1, sostituiamo ogni produzione del tipo $Aj \rightarrow Ah\beta$ con l'insieme delle produzioni $Aj \rightarrow \gamma\beta$ per ogni produzione del tipo $Ah \rightarrow \gamma$ (facendo riferimento alle produzioni di Ah già modificate);
   b) facendo uso della tecnica descritta in precedenza, rimuoviamo le eventuali ricorsioni sinistre immediate a partire da Aj (i nuovi non terminali che eventualmente vengono introdotti in questo passo non sono inseriti nella lista ordinata).

L'algoritmo appena descritto garantisce l'eliminazione di tutte le ricorsioni sinistre se si assume non solo che la grammatica non contenga λ-produzioni ma che non contenga nemmeno cicli, ovvero non sia possibile derivare il simbolo non terminale A a partire da A stesso.

Backtracking

Un modo per sviluppare un parser top-down consiste semplicemente nel fare in modo che il parser tenti esaustivamente tutte le produzioni applicabili fino a trovare l'albero di derivazione corretto. Questo è talvolta detto il

metodo della "forza bruta". Tale metodo può far sorgere tuttavia alcuni problemi. Per esempio, consideriamo la grammatica seguente: S → ee; S → bAc; S → bAe; A → d; A → cA; e la sequenza di simboli terminali bcde. Se il parser tenta tutte le produzioni esaustivamente, incomincerà considerando S → bAc poiché il b all'inizio dell'input impedisce di usare la produzione S → ee. Il prossimo simbolo di input è c: questo elimina A → d e quindi viene tentata A → cA. Proseguendo in questo modo, il parser genera un albero errato. Esso genera la stringa bcdc invece di bcde. Chiaramente ciò è dovuto al fatto che il parser è partito con la parte destra sbagliata: se fosse stato capace di guardare in avanti al simbolo di input finale, non avrebbe fatto quest'errore. Purtroppo, i parser top-down scandiscono i simboli terminali da sinistra verso destra. Per rimediare al danno, dobbiamo tornare indietro (in inglese backtrack) fino a trovare una produzione alternativa: in questo caso, il parser deve ritornare alla radice e tentare la terza produzione S → bAe.

Il backtrack è sostanzialmente una visita in profondità di un grafo. La ricerca procede in avanti da nodo a nodo finché viene trovata la soluzione oppure viene raggiunto un vicolo cieco. Se questo è il caso, deve tornare indietro fino a trovare una biforcazione lungo la strada e tentare quella possibilità. Il problema è in parte nello sviluppo del parser e in parte nello sviluppo del linguaggio. L'esempio che abbiamo usato aveva una produzione che sembrava promettente ma che aveva una trappola alla fine. Osserviamo quanti meno problemi avremmo con la seguente grammatica: S → ee; S → bAQ; Q → c; Q → e; A → d; A → cA. In questo caso abbiamo fattorizzato il prefisso comune bA e usato un nuovo non terminale per permettere la scelta finale tra c ed e. Questa grammatica genera lo stesso linguaggio di quella precedente ma il parser può ora generare l'albero di derivazione giusto senza backtrack. La trasformazione appena mostrata è nota come fattorizzazione sinistra ed è la prima di tante tecniche che rendono i parser top-down praticabili: un altro importante modo di evitare il backtrack è quello di cui parleremo nel prossimo paragrafo.

Parser predittivi

Il backtrack potrebbe essere evitato se il parser avesse la capacità di guardare avanti nella grammatica in modo da anticipare quali simboli terminali sono derivabili (mediante derivazioni sinistre) da ciascuno dei vari simboli non terminali nelle parti destre delle produzioni.

Possiamo definire un algoritmo per il calcolo della funzione FIRST distinguendo i seguenti due casi.

1. $\alpha$ è un singolo carattere oppure $\alpha = \lambda$.
   a) Se $\alpha$ è un terminale y allora FIRST($\alpha$) = y.
   b) Se $\alpha = \lambda$ allora FIRST($\alpha$) = {$\lambda$}.
   c) Se $\alpha$ è un non terminale A e A → $\beta$i sono le produzioni a partire da A, per i = 1, . . . , k, allora FIRST($\alpha$) = Unione FIRST($\beta$k).
2. $\alpha$ = X1X2 · · · Xn con n > 1.
   a) Poniamo FIRST($\alpha$) = Ø e j = 1.
   b) Poniamo FIRST($\alpha$) = FIRST($\alpha$) ∪ FIRST(Xj) − {$\lambda$}.
   c) Se Xj è annullabile e j < n, allora poniamo j = j + 1 e ripetiamo il passo precedente.
   d) Se j = n e Xn è annullabile, allora includiamo $\lambda$ in FIRST($\alpha$).

I parser che usano gli insiemi FIRST sono detti parser predittivi, in quanto hanno la capacità di vedere in avanti e prevedere che il primo passo di una derivazione ci darà prima o poi un certo simbolo terminale.

Grammatiche LL(1)

La tecnica basata sul calcolo della funzione FIRST non sempre può essere utilizzata. Talvolta la struttura della grammatica è tale che il simbolo terminale successivo non ci dice quale parte destra utilizzare (ad esempio, nel caso in cui FIRST($\alpha$) e FIRST($\beta$) non siano disgiunti). Inoltre, quando le grammatiche acquisiscono $\lambda$-produzioni come risultato della rimozione della ricorsione sinistra, gli insiemi FIRST non ci dicono quando scegliere A → $\lambda$. Per gestire questi casi, abbiamo bisogno di una seconda funzione, FOLLOW. Nel definire tale funzione, assumeremo che una sequenza di simboli terminali, prima di essere passata al parser, abbia un segno di demarcazione appeso, che indicheremo con $.

Data una grammatica G = (V, N, S, P), definiamo la funzione FOLLOW: N → $2^V$ come segue: se A è un simbolo non terminale, allora, per ogni x ∈ V che può seguire A in una forma sentenziale, x appartiene a FOLLOW(A). Per convenzione, inoltre, assumiamo che se A può apparire come ultimo simbolo di una forma sentenziale, allora $ ∈ FOLLOW(A). Per ogni non terminale A, possiamo calcolare FOLLOW(A) nel modo seguente.

1. Se A è il simbolo iniziale, allora includiamo $ in FOLLOW(A).
2. Cerchiamo attraverso la grammatica le occorrenze di A nelle parti destre delle produzioni. Sia Q → xAy una di queste produzioni. Distinguiamo i seguenti tre casi:
   a) se y inizia con un terminale q, allora includiamo q in FOLLOW(A);
   b) se y non inizia con un terminale, allora includiamo FIRST(y) − {$\lambda$} in FOLLOW(A);
   c) se y = $\lambda$ (ovvero A è in fondo) oppure se y è annullabile, allora includiamo FOLLOW(Q) in FOLLOW(A).

In un parser predittivo, l'insieme FOLLOW ci dice quando usare le λ-produzioni. Supponiamo di dover espandere un non terminale A. Inizialmente vediamo se il simbolo terminale in arrivo appartiene all'insieme FIRST di una parte destra di una produzione la cui parte sinistra è A. Se così non è questo normalmente vuol dire che si è verificato un errore. Ma se una delle produzioni da A è A → λ, allora dobbiamo vedere se il simbolo terminale appartiene a FOLLOW(A). Se così è, allora può non trattarsi di un errore e A → λ è la produzione scelta.

Grammatiche per cui questa tecnica può essere usata sono note come grammatiche LL(1) e i parser che usano questa tecnica sono detti parser LL(1). In questa notazione, la prima L sta per "left" per indicare che la scansione è da sinistra a destra, la seconda L sta per "left" per indicare che la derivazione è sinistra, e '(1)' indica che si guarda in avanti di un carattere. Le grammatiche LL(1) assicurano che guardando un carattere in avanti il token in arrivo si determina univocamente quale parte destra scegliere. Perché una grammatica sia LL(1) richiediamo che per ogni coppia di produzioni A → α e A → β, valgano le seguenti due proprietà:

1. FIRST(α) − {λ} e FIRST(β) − {λ} siano disgiunti;
2. se α è annullabile, allora FIRST(β) e FOLLOW(A) devono essere disgiunti.

Costruzione di un parser predittivo

Se possiamo evitare il backtrack, allora vi sono diversi modi di implementare il parser. Una forma più conveniente di parser predittivo consiste di una semplice procedura di controllo che utilizza una tabella. Parte dell'attrattiva di questo approccio è che la procedura di controllo è generale: se la grammatica va cambiata, solo la tabella deve essere riscritta creando meno problemi che la riprogrammazione. La tabella può essere costruita a mano per piccole grammatiche o mediante il calcolatore per grammatiche grandi. La tabella della versione non ricorsiva dice quale parte destra scegliere e i simboli terminali sono usati nel modo naturale.

Il nostro esempio sarà un parser per le espressioni aritmetiche, usando la grammatica vista nell'esempio precedente, ovvero la grammatica delle espressioni aritmetiche ottenuta dopo aver avere eliminato le ricorsioni sinistre:

E → TQ; Q → +TQ; Q → −TQ; Q → λ; T → FR; R → ∗FR; R → /FR; R → λ; F → (E); F → id

La tabella per questa grammatica è la seguente (gli spazi bianchi indicano condizioni di errore):

| | **id** | + | - | * | / | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|
| E | TQ | | | | | TQ | | |
| Q | | +TQ | −TQ | | | | ϵ | ϵ |
| T | FR | | | | | FR | | |
| R | | ϵ | ϵ | ∗FR | /FR | | ϵ | ϵ |
| F | **id** | | | | | (E) | | |

Intuitivamente, ogni elemento non vuoto della tabella indica quale produzione debba essere scelta dal parser trovandosi a dover espandere il simbolo non terminale che etichetta la riga della tabella e leggendo in input il simbolo terminale che etichetta la colonna della tabella. Pertanto, nel caso dei parser guidati da una tabella siffatta, è utile mantenere una pila che viene usata nel modo seguente:

- Inseriamo $ nella pila e alla fine della sequenza e inseriamo il simbolo iniziale nella pila.
- Fintantoché la pila non è vuota
  - Sia x l'elemento in cima alla pila e a il simbolo terminale in input.
  - Se x ∈ V allora:
    - se x = a allora estraiamo x dalla pila e avanziamo di un simbolo terminale, altrimenti segnaliamo un errore.
  - Se x /∈ V, allora:
    - se table[x, a] non è vuoto, allora estraiamo x dalla pila e inseriamo table[x, a] nella pila in ordine inverso, altrimenti segnaliamo un errore.

Notiamo che, nell'ultimo caso, inseriamo la parte destra di una produzione in ordine inverso in modo che il simbolo più a sinistra sarà in cima alla pila pronto per essere eventualmente espanso o cancellato. Ciò è dovuto al fatto che il parser sta cercando di generare una derivazione sinistra, in cui il simbolo non terminale più a sinistra è quello da espandere e al fatto che la pila è una struttura dati che consente l'inserimento e l'estrazione di un elemento dallo stesso punto di accesso. Mostriamo ora un esempio

Supponiamo che la sequenza in input sia **(id+id)*id**. Lo stato iniziale sarà il seguente:

| PILA | INPUT | PRODUZIONE | DERIVAZIONE |
|---|---|---|---|
| $E | **(id+id)*id**$ | | |

La sequenza ha il simbolo $ appeso in fondo e la pila ha tale simbolo e il simbolo iniziale inseriti (scriviamo il contenuto della pila in modo che cresca da sinistra verso destra). A questo punto l'analizzatore entra nel ciclo principale. Il simbolo in cima alla pila è E e table[E, (] = TQ: quindi la nostra produzione è E → TQ e abbiamo

| PILA | INPUT | PRODUZIONE | DERIVAZIONE |
|---|---|---|---|
| $E | **(id+id)*id**$ | E → TQ | E → TQ |
| $QT | **(id+id)*id**$ | | |

In questo caso, E è stato estratto dalla pila e la parte destra TQ inserita nella pila in ordine inverso. Ora abbiamo un non terminale T in cima alla pila e il simbolo terminale in input è ancora (. Quindi abbiamo table[T, (] = FR e la produzione è T → FR:

| PILA | INPUT | PRODUZIONE | DERIVAZIONE |
|---|---|---|---|
| $E | **(id+id)*id**$ | E → TQ | E → TQ |
| $QT | **(id+id)*id**$ | T → FR | → FRQ |
| $QRF | **(id+id)*id**$ | | |

Proseguendo abbiamo table[F, (] = (E) e la produzione è F → (E):

| PILA | INPUT | PRODUZIONE | DERIVAZIONE |
|---|---|---|---|
| $E | **(id+id)*id**$ | E → TQ | E → TQ |
| $QT | **(id+id)*id**$ | T → FR | → FRQ |
| $QRF | **(id+id)*id**$ | F → (E) | → (E)RQ |
| $QR)E( | **(id+id)*id**$ | | |

Ora abbiamo un terminale in cima alla pila. Lo confrontiamo con il simbolo in input e, poichè coincidono, estraiamo il simbolo terminale dalla pila e ci spostiamo al prossimo simbolo della sequenza in input:

| PILA | INPUT | PRODUZIONE | DERIVAZIONE |
|---|---|---|---|
| $E | **(id+id)*id**$ | E → TQ | E → TQ |
| $QT | **(id+id)*id**$ | T → FR | → FRQ |
| $QRF | **(id+id)*id**$ | F → (E) | → (E)RQ |
| $QR)E( | **(id+id)*id**$ | | |
| $QR)E | **id+id)*id**$ | | |

Così andando avanti si arriva a consumare l'intero input con la pila vuota e possiamo annunciare il successo

Costruzione della tabella del parser predittivo

Ricordiamo che in un parser top-down, se non vi deve essere backtrack, allora il simbolo terminale in arrivo deve sempre dirci cosa fare. La stessa cosa si applica alla costruzione della tabella. Supponiamo che abbiamo X in cima alla pila e che a sia il simbolo terminale in arrivo. Vogliamo selezionare una parte destra che incominci con a oppure che possa portare a una forma sentenziale che inizi con a.

Per esempio, all'inizio del nostro esempio, avevamo E nella pila e ( come input. Avevamo bisogno di una produzione della forma E → ( ⋯ Ma una tale produzione non esiste nella grammatica. Poiché non era disponibile, avremmo dovuto tracciare un cammino di derivazione che ci conducesse a una forma sentenziale che iniziasse con (. L'unico tale cammino è E → TQ → FRQ → (E)RQ e se guardiamo alla tabella vediamo che essa contiene esattamente la parte destra che determina il primo passo del cammino.

Ciò ci sta conducendo verso un terreno familiare: vogliamo selezionare una parte destra α se il token appartiene a FIRST(α); quindi per una riga A e una produzione A → α, la tabella deve avere la parte destra α in ogni colonna etichettata con un terminale in FIRST(α). Ciò funzionerà in tutti i casi eccetto quello in cui FIRST(α) include λ poiché la tabella non ha una colonna etichettata λ. Per questi casi, seguiamo gli insiemi FOLLOW.

La regola per costruire la tabella è dunque la seguente.

- Visita tutte le produzioni. Sia X → β una di esse.
  - Per tutti i terminali a in FIRST(β), poniamo table[X, a] = β.
  - Se FIRST(β) include λ, allora, per ogni a ∈ FOLLOW(X), table[X, a] = λ.

Nel caso della grammatica delle espressioni matematiche, usando gli insiemi FIRST e FOLLOW precedentemente calcolati e le regole sopra descritte, otteniamo la tabella mostrata in precedenza. Infine, se la grammatica è di tipo LL(1) siamo sicuri che la tabella non conterrà elementi multipli.

Parser bottom-up

Generano derivazioni destre e costruiscono l'albero di derivazione partendo dalle foglie (in post-ordine). Si utilizza quindi l'analisi predittiva shift-reduce basata su grammatiche LR(k). Si leggono i token dall'input (shift) e si inseriscono in

## semplice esempio

linguaggio non-contestuale
$L = \{a^n b^m c^{n+m} \mid n, m > 0\}$

produzioni

$S \rightarrow aSc \mid aTc$

$T \rightarrow bTc \mid bc$

esempio generazione *abbbcccc*

$\underline{S} \Rightarrow a\underline{Tc} \Rightarrow ab\underline{Tcc} \Rightarrow abb\underline{Tccc} \Rightarrow abbbcccc$

simbolo convenzionale di fine input: $

| | pila | undigested | azione |
|---|---|---|---|
| 1 | | *abbbcccc*$ | shift |
| 2 | *a* | *bbbcccc*$ | shift |
| 3 | *ab* | *bbcccc*$ | shift |
| 4 | *abb* | *bcccc*$ | shift |
| 5 | *abbb* | *cccc*$ | shift |
| 6 | *abbbc* | *ccc*$ | reduce (*T→bc*) |
| 7 | *abbT* | *ccc*$ | shift |
| 8 | *abbTc* | *cc*$ | reduce (*T→bTc*) |
| 9 | *abT* | *cc*$ | shift |
| 10 | *abTc* | *c*$ | reduce (*T→bTc*) |
| 11 | *aT* | *c*$ | shift |
| 12 | *aTc* | $ | reduce (*S→aTc*) |
| 13 | *S* | $ | accetta |

una pila tentando di costruire sequenze da ridurre con un non terminale (reduce). Se l'analisi va a buon fine si consuma tutto l'input ottenendo solo il simbolo iniziale nella pila. Se ad un certo punto non è più possibile operare uno shift o una reduce si ottiene un errore. L'input è diviso in undigested (da leggere) e semi-digested (inserito in pila). Se in pila abbiamo una stringa $\alpha$ affiorante ed esiste una produzione X→$\alpha$ si esegue pop di tutti i caratteri di $\alpha$ e push di X ($\alpha$ è detta handle).

Problemi: esistono due tipi di conflitti

- Reduce-reduce. Sono rari e causati da errori nella costruzione della grammatica
- Shift-reduce (esempio del dangling else – else appeso). L'ambiguità viene risolta con regole aggiuntive che non appaiono nella grammatica

LR parsing

Generalmente la produzione da applicare dipende sia dall'handle che dal contesto, ovvero dagli altri simboli presenti in pila. Per ovviare a questo problema si inseriscono nella pila degli stati che rappresentano il contesto. Lo stato affiorante in pila consente di prendere la decisione corretta. Si costruiscono delle tavole: action e goto. Action[s,a] descrive quale azione eseguire quando lo stato affiorante in pila è s e viene letto a in input. Le possibili operazioni sono shift, reduce, accept, report error. Goto[s,X] indica il nuovo stato s da inserire in cima alla pila dopo la riduzione del non terminale X.

## LR parsing in azione

Grammatica
1) E –> E + T  2) E –> T  3) T –> (E)  4) T –> id
Input id+(id)

Tavole Action e GOTO insieme

| Stato in cima stack | Action | | | | | GOTO | |
|---|---|---|---|---|---|---|---|
| | id | + | ( | ) | $ | E | T |
| 0 | s4 | | s3 | | | 1 | 2 |
| 1 | | s5 | | | accetta | | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s4 | | s3 | | | 6 | 2 |
| 4 | r4 | r4 | r4 | r4 | r4 | | |
| 5 | s4 | | s3 | | | | 8 |
| 6 | | s5 | | s7 | | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | r1 | r1 | r1 | r1 | r1 | | |

analisi di id+(id)

| PILA | INPUT | Azioni |
|---|---|---|
| s0 | id + (id)$ | Shift: s4 in pila; avanza in input |
| s0 s4 | + (id)$ | Reduce: 4) T –> id; pop id da pila; goto s2; input non cambia |
| s0 s2 | + (id)$ | Reduce: 2) E –> T; goto s1 |
| s0 s1 | + (id)$ | Shift: s5 in pila; avanza in input |

Notazione punto

In alcuni casi posso trovare un handle ma potrebbe non essere la scelta corretta. Per ovviare a questo problema si introduce la notazione punto. Si mette un punto per separare la parte destra di una produzione in due sottosequenze: a sinistra del punto abbiamo gli elementi già letti ed impilati, a destra gli elementi ancora da analizzare. Ad esempio E→E•+T è un item. Si aggiunge una produzione S'→S per avere un assioma che non compaia in parti destre. Per ogni produzione si considerano tutti gli item possibili (mettendo il puntino prima, in mezzo e dopo). Lo stato di un parser diventa quindi la conseguenza della chiusura (collezione di più item). Per ogni item A→a•Bb si aggiunge al set B→•c.

## determinazione altri stati

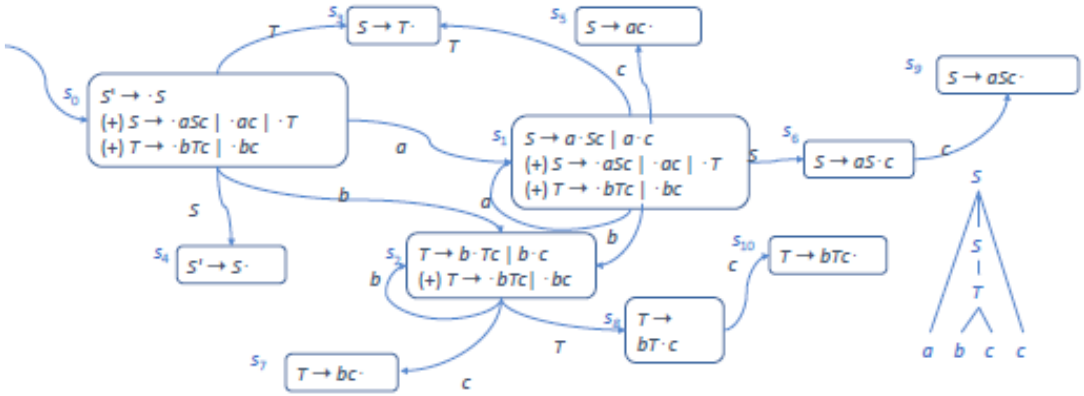
$S' \rightarrow S$
$S \rightarrow aSc \mid ac \mid T$
$T \rightarrow bc \mid bTc$

stato $s_0$
$S' \rightarrow \cdot S$
(+) $S \rightarrow \cdot aSc \mid \cdot T \mid \cdot ac$
(+) $T \rightarrow \cdot bTc \mid \cdot bc$

- da $s_0$ a $s_1$ (simbolo *a*)
  $S \rightarrow a \cdot Sc \mid a \cdot c$
  (+) $S \rightarrow \cdot aSc \mid \cdot ac \mid \cdot T$
  (+) $T \rightarrow \cdot bTc \mid \cdot bc$
- da $s_0$ a $s_2$ (simbolo b)
  $T \rightarrow b \cdot Tc \mid b \cdot c$
  (+) $T \rightarrow \cdot bTc \mid \cdot bc$
- da $s_0$ a $s_3$ (simbolo *T*)
  $S' \rightarrow T \cdot$
- da $s_0$ a $s_4$ (simbolo *S*)
  $S' \rightarrow S \cdot$

# goto-graph (transition diagram)

$S' \rightarrow S$
$S \rightarrow aSc \mid ac \mid T$
$T \rightarrow bc \mid bTc$




| stato | a | b | c | $ | S | T |
|---|---|---|---|---|---|---|
| 0 | shift 1 | shift 2 | | | goto 4 | goto 3 |
| 1 | shift 1 | shift 2 | shift 5 | | goto 6 | goto 3 |
| 2 | | shift 2 | shift 7 | | | goto 8 |
| 3 | reduce $S \rightarrow T$ | | | | | |
| 4 | | | | accept | | |
| 5 | reduce $S \rightarrow ac$ | | | | | |
| 6 | | | shift 9 | | | |
| 7 | reduce $T \rightarrow bc$ | | | | | |
| 8 | | | shift 10 | | | |
| 9 | reduce $S \rightarrow aSc$ | | | | | |
| 10 | reduce $T \rightarrow bTc$ | | | | | |
| | Tavola Action | | | | Tavola Goto | |

Una grammatica è LR(0) se per ogni stato è presente al più una produzione con • finale e non sono presenti contemporaneamente una produzione con il punto finale e una con il punto non finale. La decisione shift o reduce viene presa senza guardare all'input undigested. Rimangono da gestire i conflitti shift/reduce e reduce/reduce. Guardando k>=1 caratteri in input è possibile effettuare LR parsing su molte più grammatiche. Tutti i linguaggi context free deterministici ammettono una grammatica generatrice LR(1).

Algoritmo per generare tabelle action e goto:

1. Costruire insieme di stati
   a. Inizia con lo stato iniziale, prendi le produzioni dall'assioma e considera la sua chiusura
   b. Determina il prossimo stato esaminando le possibili produzioni, esegui la chiusura e determina un nuovo stato
   c. Itera il passo precedente fino ad aver esaminato tutte le transizioni.
2. Costruisci tabelle Action e Goto