Linguaggi di programmazione

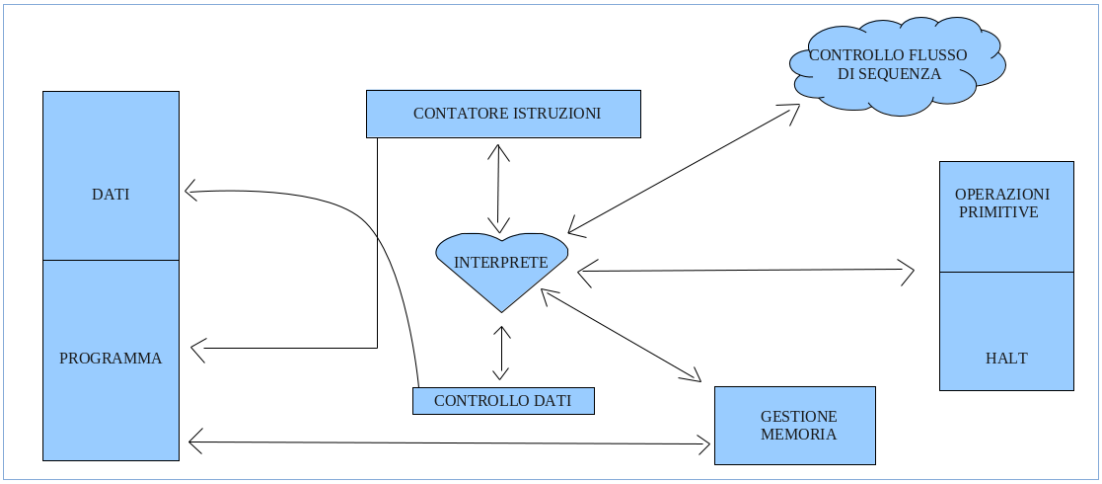
Mele Giampaolo
con la revisione di Giuliano Vallese

April 14, 2011

16/03/2011

Macchine astratte

In informatica il termine macchina astratta indica un modello teorico di hardware o software, in grado di eseguire operazioni, memorizzarne il risultato e seguire il flusso dell'algoritmo.
La macchina astratta come ben ci aspettiamo è quindi un modello per descrivere una macchina reale. Considereremo d'ora in avanti che la nostra macchina astratta sia di tipo sequenziale, ovvero sia in grado di fare una sola operazione la volta, descriviamo ora come funziona una tale macchina



Teniamo a mente il diagramma disegnato sopra e spieghiamo come funziona una macchina astratta con un esempio. Supponiamo per semplicità che l'interprete sia un essere umano che non sappia risolvere le equazioni ma che sappia fare solo delle "operazioni primitive" tra numeri (somma, prodotto, sottrazione, divisione) e supponiamo che il suo compito sia di risolvere

$$\frac{2 \cdot 7}{x} + (3 - x)$$

Il "programma" contiene la procedura per risolvere le equazioni (ad esempio nel nostro caso semplificato dove l'interprete è un esser umano, il programma lo si può pensare come un libro di algebra).
Il contatore istruzioni servirà a far eseguire nell'ordine sequenziale giusto, ad esempio facciamo prima il prodotto, dopo la divisione e dopo la somma,... .
Il controllo dati serve per controllare la coerenza dei dati (se ad esempio chiedo di sommare il carattere A con il carattere B non sono in grado di farlo dato che l'operazione primitiva che conosco è tra numeri).
La gestione della memoria serve per gestire lo spazio (nel nostro caso se l'interprete ha due fogli non possiamo chiedere che risolva un'equazione troppo lunga per cui sono necessari 3 fogli, o almeno la gestione della memoria deve esser in grado di cancellare le zone di memoria che non servono e riutilizzarle).
La zona dati ha un significato ovvio, contiene i dati del problema da risolvere (notare che dati e programma sono accorpati, infatti sono sempre nella memoria della nostra macchina, come se il libro di algebra avesse delle pagine bianche per risolvere gli esercizi).
Il controllo flusso di sequenza serve per controllare che l'interprete stia facendo tutto bene.
Inoltre tra le operazioni primitive (che appunto in questo caso sono somma, sottrazione, prodotto e divisione), abbiamo inserito anche "Halt", ovvero il nostro interprete è in grado di fermarsi quando ha il risultato (quando ha risolto l'equazione o è riuscito a stabilire che non può farlo per motivi di memoria o incoerenza dati).

A questo punto l'interprete può risolvere il problema che gli è stato posto lavorando come una macchina sequenziale astratta (il concetto di interprete è molto importante, possiamo pensarlo come il cuore di una macchina astratta).

Ricapitolazione

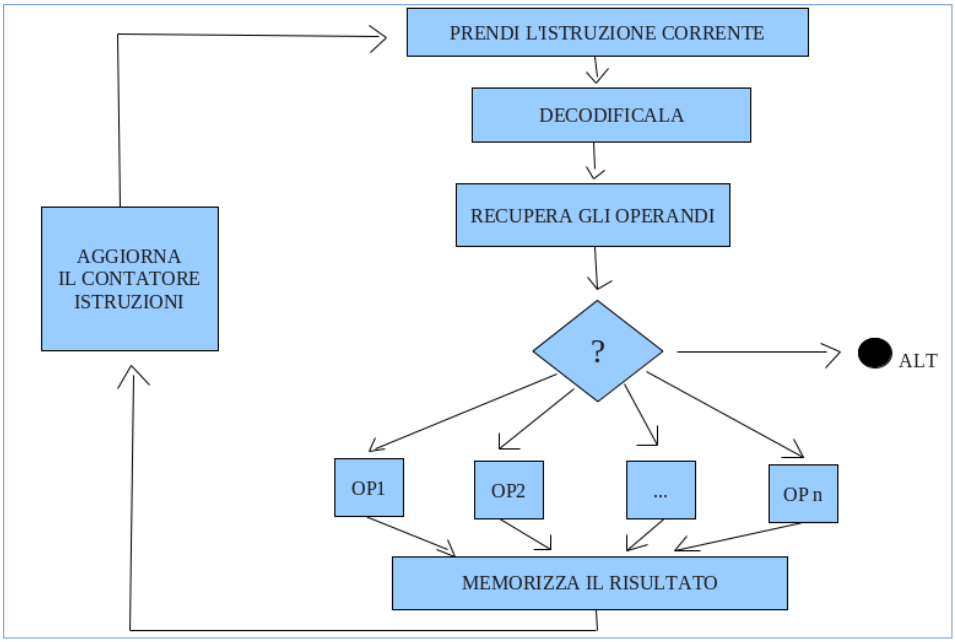
Ricapitolando abbiamo:

- Operazioni primitive e dati primitivi
- Operazioni e dati per il controllo di sequenza
- Operazioni e dati per la gestione dei dati
- Operazioni e dati per la gestione di memoria

Come conseguenza tutte le macchine astratte sequenziali hanno la medesima struttura e in un senso che specificheremo dopo, tutti i linguaggi sequenziali sono uguali.

Interprete

Studiamo in dettaglio cosa fa l'interprete, il tutto è riassunto dal seguente diagramma:



Quindi l'interprete prenderà l'istruzione corrente (ad esempio eseguire il prodotto $2 \cdot 7$), la decodificherà (ad esempio se l'interprete sa fare solo operazioni tra numeri binari, è necessario decodificare i numeri in binario), recuperare gli operandi ed eseguire una delle operazioni primitive che sa fare tra $OP1, OP2, \ldots, OPn$ (in questo caso il prodotto), oppure fermarsi se si è alla fine del programma o c'è un'incoerenza nei dati (Halt), memorizzare il risultato (se necessario ricodificare in decimale) e infine aggiornare il contatore istruzioni, poi si ricomincia da capo.

Osservazione 1
Il diagramma è di tipo ciclico (come il while)

Osservazione 2
Quello che sto facendo è cercare un punto fisso (per fermarmi)

Prima si era accennato che i linguaggi sequenziali sono in qualche senso tutti uguali, torneremo su questo aspetto ma osserviamo, dai linguaggi che conosciamo, che quello che si fa con un linguaggio lo si può fare con tutti, quello che cambia sono le operazioni, i tipi di dati e i modi per combinare dati e operazioni tra di loro.

Esempio 1
Supponiamo di avere due linguaggi che hanno le stesse operazioni primitive e gli stessi tipi di dati, ma uno opera su $\mathbb{N}$ e uno opera su $\mathbb{Q}$, allora i linguaggi

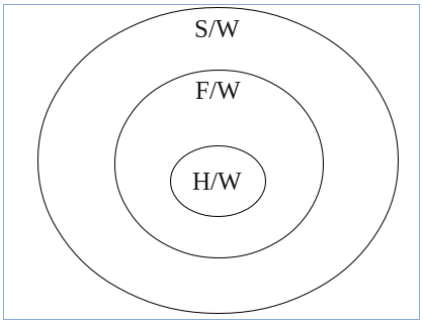
sono equivalenti dato che i due insiemi sono in bigezione.

Esempio 2
L'operatore -case- e l'operatore -if- sono equivalenti, quindi due linguaggi che operano sullo stesso insieme, con lo stesso tipo di dati e con le stesse operazioni ma uno con il -cases- e l'altro con l'-if- sono equivalenti.

Realizzazione di una macchina astratta

Per ora abbiamo definito tutto in astratto ma in concreto come realizziazi-amo tutto ciò partendo da cose tangibili?

Abbiamo un nucleo detto hardware, questo è il centro di tutto, è in grado di fare pochissime operazioni elementari usando fenomeni fisici (ad esempio un processore fa operazioni in binario sfruttando la dualità dello stato di un filo elettrico, ovvero 'passa corrente'/'non passa corrente').
Il firmware invece è il livello successivo, permette di fare operazioni più complesse usando le operazioni elementari dell'hardware (ad esempio se so fare le operazioni tra numeri binari allora le so fare tra numeri razionali, o anche se so fare somme so fare prodotti, elevamenti a potenze e quindi so calcolare le funzioni razionali).
Il software fa fondamentalmente tutto il resto.



Chiameremo programma, l'implementazione di un algoritmo in un linguaggio adatto a essere 'compreso' ed eseguito da un computer o da una macchina virtuale.

Non abbiamo ancora dato una definizione di linguaggio ma per ora chiamiamo "semantica del linguaggio" la funzione che associa alla coppia (programma,dati) l'elemento (dati).
Ad esempio se il programma fa la somma di due numeri e moltiplica per due, ovvero

$$programma(a, b) = 2\ (a + b)$$

allora la semantica del linguaggio associa alla coppia

$$(\text{programma}, (a, b)) \rightarrow (\text{programma}(a, b))$$

in generale la semantica del linguaggio sono le funzioni

$$\text{PROGRAMMA} \times \text{DATI} \longrightarrow \text{DATI}$$

$$((\text{programma,dati})) \longrightarrow \text{programma(dati)}$$

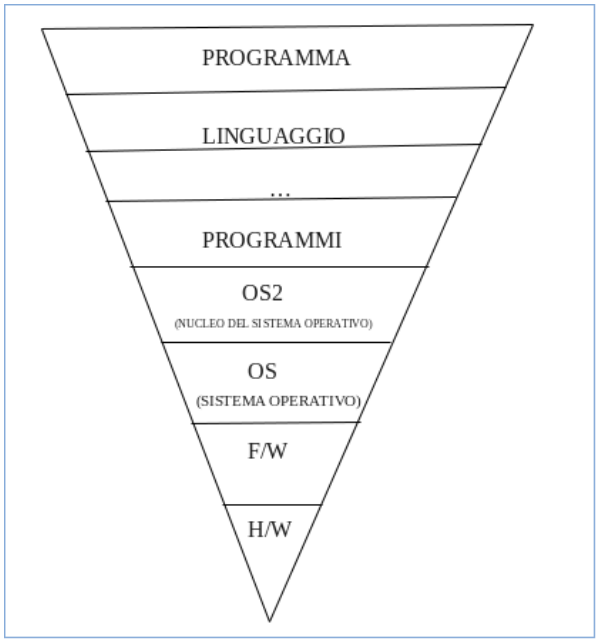
Il programma lo esegue l'interprete della macchina astratta.

Posso inoltre considerare le funzioni del tipo

$$\text{PROGRAMMA} \longrightarrow (\text{DATI} \longrightarrow \text{DATI})$$

ovvero associo ad ogni programma una funzione che va da DATI a DATI detta "eseguibile", ad esempio dal programma mostrato prima l'eseguibile associerà al dato (a, b) il dato 2 $(a + b)$.

Nella figura che segue è mostrato come una macchina lavori a livelli come spiegato fin ora



21/03/2011

Riepilogo lezione precedente

Abbiamo dato l'idea intuitiva di linguaggio, questo serve a descrivere i problemi ma anche a risolverli, abbiamo inoltre associato ad un linguaggio una

macchina astratta e a questa abbiamo associato un interprete.

$$\text{LINGUAGGIO} \rightarrow \text{MACCHINA ASTRATTA} \rightarrow \text{INTERPRETE}$$

Come abbiamo anticipato la volta scorsa tutti i linguaggi sequenziali sono equivalenti in un qualche senso, ovvero se hanno le stesse strutture, le stesse opearazioni tra strutture e le stesse operazione primitive allora è possibile fare le stesse cose. Ma ben sappiamo che ci sono linguaggi più adatti di altri in base a quello che si deve fare, ad esempio per scrivere una pagina HTML useremo il linguaggio HTML, per risolvere problemi matematici useremo Matlab o in altre circostanze può esser comodo usare Python o altri.
Cercheremo di capire inoltre come dato un programma e dei dati, ottenere il risultato dell'esecuzione.
Si è anche definita la semantica del linguaggio come

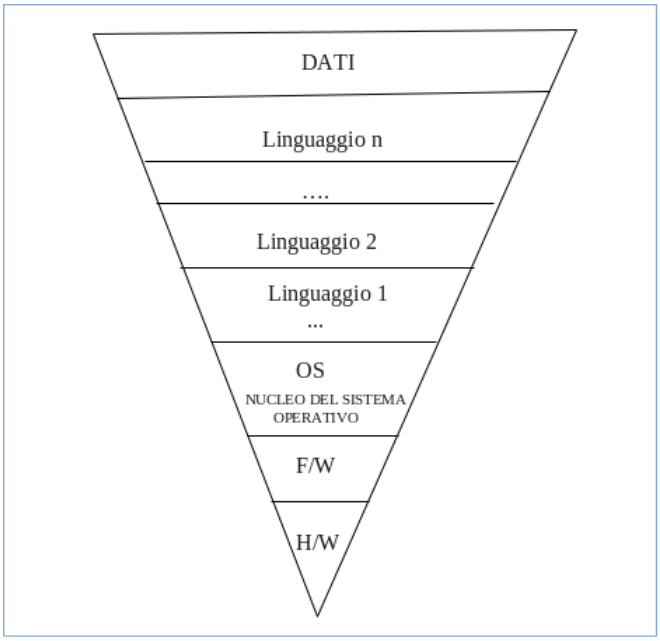
$$\text{PROGRAMMA} \times \text{DATI} \rightarrow \text{DATI}$$

e si è detto che il programma deve eseguirlo l'interprete seguendo il diagramma disegnato nella lezione precedente.

Inoltre si è anche definito il compilatore come l'applicazione che associa ad un programma il suo eseguibile che chiameremo programma oggetto.

$$\text{PROGRAMMA} \rightarrow (\text{DATI} \rightarrow \text{DATI})$$

Abbiamo inoltre visto la struttura con cui funziona una macchina reale



Anche se per ora può non sembrare intuitivo, da questa struttura si trova che non c'è differenza tra dati e linguaggio dato che il linguaggio L_i prende in ingresso il linguaggio L_j, ad esempio quando scriviamo un programma in C e lo compiliamo quello che succede è che il tutto viene tradotto in un linguaggio che la macchina può capire meglio, ma questo si vede bene più avanti.
Ad ogni modo, il fatto che scompaia la differenza tra linguaggio e dati ci crea un prosso problema: fa nascere la necessità di avere un insieme in cui vivono i dati che deve avere la proprietà DATI $\simeq$ (DATI $\rightarrow$ DATI) e per questione di cardinalità cio non è possible. Il problema è stato risolto ma è troppo complicato vedere come, basti sapere che esiste un insieme detto D_∞ con la proprietà $D_\infty \simeq (D_\infty \rightarrow D_\infty)$.

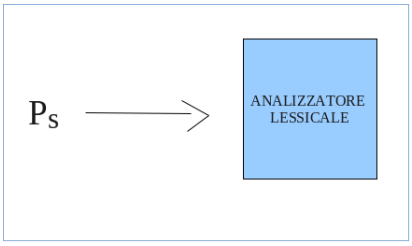
Esecuzione di un programma

Come anticipato prima, eseguire un programma non è così immediato per una macchina, infatti cio che è immediato per un essere umano non lo è per una macchina e viceversa. Mostriamolo con un esempio. D'ora in avanti chiamaremo P_s il codice sorgente del programma, ad esempio supponiamo che il nostro P_s sia il seguente

```
pippo:=pigreco+2;
IF  pippo<0 THEN  ...
```

L'esecuzione di un tale programma, scritto così, è difficile per una macchina, infatti abbiamo associato un nome alle variabili, pigreco è infatti un numero e pippo è anche un numero ma la stringa "pippo" contiene 5 caratteri mentre la stringa "pigreco" ne contiene 7, quindi già questa è una difficoltà, quindi il codice sorgente deve esser "tradotto" in modo da ricompattare il tutto e fare in modo che variabili dello stesso tipo usino la stessa memoria.

Il primo passo sarà dunque quello, ricompattare il tutto per far occupare medesima memoria allo stesso



Quello che farò sarà dunque creare una tabella detta tabella simboli.

PIPPO	INTERO	VARIABILE	5
PIGRECO	REALE	COSTANTE	3.14...
:=	ASSEGNAZIONE		

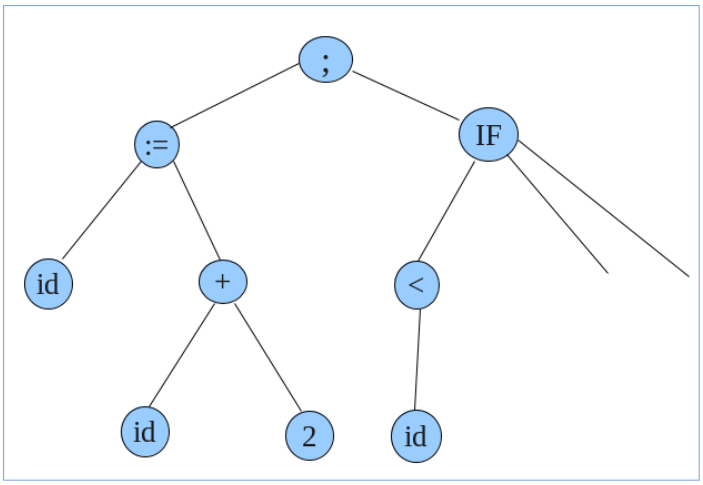
+	OPERATORE ARITMETICO		
2	REALE/INTERO	COSTANTE	2
;	DELIMITATORE		
IF	OPERATORE		
THEN	OPERATORE		

Assoceremo alle variabili dei numeri detti identificatori che permettono l'accesso alla tabella dei simboli. Possiamo ad esempio immaginare che l'identificatore associ alla variabile un numero, ad esempio la riga della tabella dei simboli in cui la variabile è contenuta. Ci sono molti modi per costruire identificatori e creare/consultare/modificare la tabella dei simboli, questa tabella infatti dovrà contenere anche le altre strutture e operazioni; seguiamo per ora l'associazione descritta per righe.
Quindi a PIPPO assoceremo il numero 1, a PIGRECO il numero 2 e così via anche con gli operatori e i delimitatori. Quindi il nostro codice sorgente sarà trasformato in una successione di numeri e caratteri che permette alla macchina di capire il programma e all'interprete di eseguirlo, in questo modo evitiamo ad esempio il problema mostrato prima sulla lunghezza delle stringhe che identificano le variabili.

Daltronde bisogna rispettare delle regole, ad esempio non possiamo fare questa assegnazione

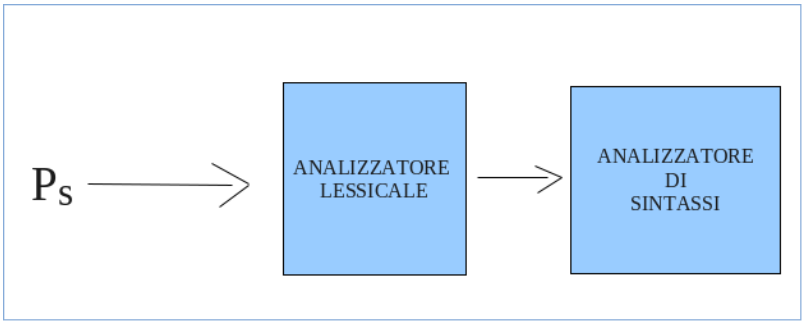
```
pippo:=IF;
```

Per risolvere questo problema devo generare un albero di sintassi, questo come radice avrà ; e poi le varie diramazioni sono il codice, nel nostro esempio

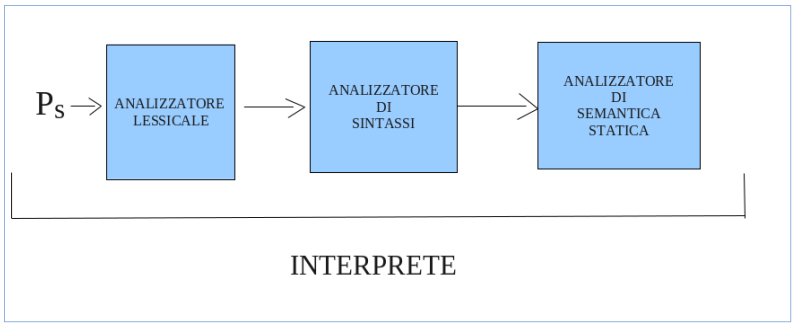


Per controllare la coerenza partiamo da ; e controlliamo che il sottoalbero sinistro sia coerente, che il sottoalbero destro sia coerente e se il raccordo con la radice è coerente (al primo passo la radice è ; poi potrebbe esser per qualche sottoalbero che la radice sia IF quindi bisognerà controllare che il raccordo sia

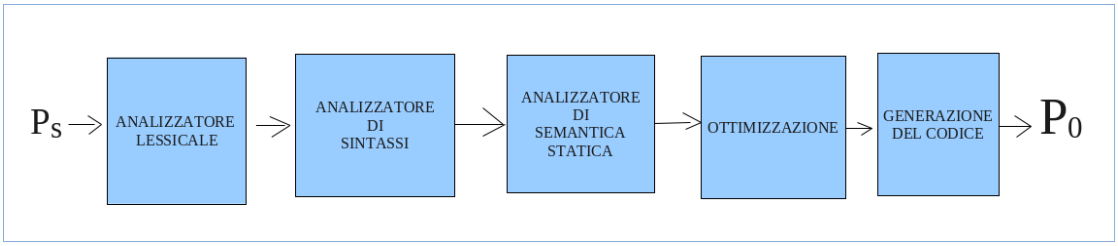
coerente, ad esempio IF a < 5 potrebbe non esserlo se l'interprete non è in grado di confrontare caratteri e numeri). Quindi procediamo con il solito algoritmo di induzione sull'albero per fare l'analisi di sintassi, quindi aggingiamo un passaggio



C'è ancora un passaggio da fare, devo controllare che a livello di tipi tutto sia coerente, ad esempio non posso sommare ad una stringa un numero a meno che il linguaggio non lo permetta. Nel nostro esempio abbiamo implicitamente sommato un numero reale (pigreco) con un numero intero 2 per ottenere un numero intero 5, ad esempio supponiamo che tra le operazioni primitive ci sia la conversione reale-intero prendendo la parte intera. Ad ogni modo per fare i controlli su i TIPI abbiamo l'analisi di semantica statica che ci dice se il nostro programma sorgante va bene.



Qui si conclude il lavoro dell'interprete, ma in generale la procedura continua



L'ottimizzazione evita che ci sia speco di tempo, ad esempio se facciamo un'assegnazione nel while inutile questa viene tirata fuori

```
k=0
  while(k<5)
```

```
a=1;
k++;
endwhile
```

In questo caso durante l'ottimizzazione l'assegnazione $a = 1$ viene tirata fuori dal while

```
k=0
a=1;
   while(k<5)
   k++;
   endwhile
```

In realtà il passo dell'ottimizzazione è molto complicato ma non è obbiettivo del corso soffermarsi su questo aspetto.
Il passo di generazione del codice invece tradurrà il nostro linguaggio in un linguaggio più complicato ma comprensibile alla macchina e infine avremo il nostro programma oggetto P_O (o eseguibile).

Calcolabilità

Una procedura per risolvere un problema è detto algoritmo e chiediamo che questo debba rispettare le seguenti

- Deve avere una rappresentazione finita (è fatto da un insieme finito di istruzioni)
- Le istruzioni sono in numero finito con effetto limitato e operano su dati discreti
- Una computazione è una successione di passi discreti
- Ogni passo dipende solo dai precedenti e da una porzione finita di dati in modo deterministico
- Non c'è limite al numero di passi e allo spazio che contiene i dati

Quindi non ci poniamo problemi di spazio e di tempo (l'algoritmo può durare anche milioni di anni ma deve finire). Questa è la definizione di algoritmo.

Macchina di Turing

Supponiamo di avere un impiegato preciso ma stupido, questo ha un foglio con delle istruzioni e della carta su cui scrivere seguendo le istruzioni. Possiamo immaginare che l'impiegato scriva su una striscia di carta piuttosto che su un foglio per semplificare il tutto.
Inoltre supponiamo che l'impiegato sia in grado di leggere un solo carattere la volta e abbia degli stati ad esempio l'insieme degli stati potrebbe essere: scrittura, lettura, ecc.

Quindi il nostro impgiegato avrà una striscia di carta piena di caratteri , lui può leggerne uno la volta e può scrivere nella posizione corrente (sul carattere che sta leggendo, quindi lo cancella) e dopo spostarsi a destra, a sinistra o restare nella stessa posizione.

Definiamo formalmente la macchina di Turing come

$$M = (Q, \Sigma, \delta, q_0)$$

Dove Q è l'insieme (finito) degli stati, Σ è l'insieme (finito) dei simboli, δ è la funzione di transizione e q_0 è lo stato iniziale.

E' necessario introdurre un ulteriore stato che non sia in Q che permetta all'impiegato di fermarsi, lo chiameremo $h \notin Q$.
Σ è detto alfabeto, anche qui è necessario introdurre un nuovo carattere # che servirà a scrive il carattere bianco (lasciare uno spazio).
Inoltre ci servirà ancora $L, R, - \notin \Sigma$ dove L vuol dire spostati di una casella a sinistra, R spostati di una casella a destra e $-$ vuol dire stai fermo sulla casella in cui sei adesso. (abbiamo implicitamente suddiviso la striscia in caselle e ogni casella contiene un carattere).
Definiamo la funzione di transizione come

$$\delta : Q \times \Sigma \to (Q \cup \{h\}) \times \Sigma \times \{L, R, -\}$$

Quindi la funzione di transizione associa ad uno stato dell'impiegato e ad un carattere (che legge dalla striscia) un'altro stato, un carattere, che scriverà sulla striscia nella posizione corrente e una direzione tra $L, R, -$ che dice all'impiegato se restare nella stessa posizione, spostarsi di una cella a sinistra o a destra.

Introduciamo anche il simbolo respingente $\Delta \notin \Sigma$ che serve per non uscire dalla memoria, ovvero se la striscia è all'inizio l'impiegato non può spostarsi a sinistra.

$$\delta(q, \Delta) = (q', a, d) \quad \Rightarrow \quad d = R \qquad a = \Delta$$

Ad esempio l'impiegato potrebbe scrivere

Δ	a	b	a	#	o	...

Esempio di macchina di Turing

q	σ	δ (q,σ)
q_0	Δ	q_0, Δ, R
q_0	$\#$	$h, \#, -$
q_0	a	$q_1, \#, L$
q_1	$\#$	$q_0, \#, L$
q_1	a	$q_0, a, -$

Questa tabella identifica univocamente la macchina di Turing.

22/03/2011

Monoide libero generato da un alfabeto

Prodotto di concatenazione

Dati due insiemi A e B definiamo l'insieme

$$A \cdot B = \{w \cdot w' \quad \text{tale che} \quad w \in A, w' \in B\}$$

Dove $\cdot$ è detto prodotto di concatenazione e ha la sola funzione di "attaccare" due simboli.

Quindi condideriamo un alfabeto Σ, definiamo induttivamente

- $\Sigma^0 = \{\epsilon\}$ dove ϵ è la stringa (/parola) vuota
- $\Sigma^{i+1} = \Sigma \cdot \Sigma^i$

Allora definiamo monoide libero generato da Σ l'insieme

$$\Sigma^\infty = \bigcup_{i \in \mathbb{N}} \Sigma^i$$

Ad esempio se prendiamo come alfabeto $\Sigma = \{0, 1\}$ allora il monoide libero generato sarà l'insieme di tutte le stringhe in 0 e 1, ovvero

$$\Sigma^\infty = \{\epsilon, 0, 1, 00, 11, 01, 10, 000, 001, 011, \cdots\}$$

Configurazione di una macchina di Turing

Ricordiamo la definizione di macchina di Turing

$$M = (Q, \Sigma, \delta, q_0)$$

Chimeremo configurazione di una macchina di Turing la quadrupla

$$(q, u, a, v)$$

Dove

- q è lo stato dell'impiegato
- $u \in \Sigma^*$ è la stringa precedente al carattere che l'impiegato sta leggendo (dall'inizio delle istruzioni)
- $a \in \Sigma$ è il carattere che l'impiegato sta leggendo
- $v \in \Sigma^*$ è la stringa che successiva da a fino alla fine delle istruzioni

In realtà v non è un qualsiasi elemento di Σ^* ma evitiamo di complicare il tutto.

D'ora in avanti più brevemente indicheremo una configurazione con $(q, u\underline{a}v)$ sottointendento il tutto e indicheremo con $\gamma = (q, u\underline{a}v)$ la configurazione.

Funzioni e insiemi

E' necessario ora introdurre la nozione di funzione come sottoinsieme del prodotto cartesiano tra insieme di partenza e insieme di arrivo.

Data una funzione $f : A \to B$ possiamo vederla come un insieme che denoteremo con la stessa lettera $f \subseteq A \times B$ che rispetta la seguente proprietà di buona definizione

$$(a, b) \in f \quad , \quad (a, c) \in f \Rightarrow b = c$$

Definiamo il dominio della funzione come

$$dom(A) = \{a \in A \ \text{ tali che } \ \exists b \in B \ \text{ che soddisfa } \ (a, b) \in f\}$$

Diremo che il dominio è il luogo in cui la funzione è definita.

Osserviamo inoltre che $\emptyset \subset A \times B$, quindi

$$\emptyset : A \to B$$

sarà una funzione, ma il suo dominio è vuoto, quindi chiameremo questa funzione indefinita.

Diremo che una funzione è finita se la cardinalità di f inteso come insieme è finita, mentre diremo che una funzione è totale se il suo dominio è tutto $\mathbb{N}$.

Osservazione
La funzione di transizione

$$\delta \subseteq Q \times \Sigma \times \{Q \cup \{h\}\} \times \Sigma \times \{R, R, -\}$$

è una funzione finita

Passo di calcolo

Intuitivamente il passo di calcolo è dato dall'esecuzione di una istruzione che l'impiegato legge e dal suo cambiamento di stato, ovvero il passaggio da una configurazione alla successiva, formalmente, posto $a, b, c \in \Sigma$ e $u, v, r \in \Sigma^*$ i possibili passi sono

1 $(q, u\underline{a}v) \to (q', u, \underline{b}, v)$ se $\delta(q, a) = (q', b, -)$

2 $(q, uc\underline{a}v) \to (q', u\underline{c}bv)$ se $\delta(q, a) = (q', b, L)$

3 $(q, u\underline{a}cv) \to (q', ub\underline{c}v)$ se $\delta(q, a) = (q', b, R)$

4 $(q, u\underline{a}) \longrightarrow (q', ub\#)$ se $\delta(q, a) = (q', b, R)$

Chiusura transitiva e riflessiva di una relazione

Data una relazione $R \subseteq A \times A$ definiamo chiusura transitiva e riflessiva di R l'insieme $R^* \subseteq A \times A$ come la più piccola relazione transitiva e riflessiva tale che $R \subseteq R^*$.

Computazione

Data una configurazione γ e un'altra configurazione γ' indicheremo con una fraccia

$$\gamma \to \gamma'$$

se con un passo di calcolo dalla configurazione γ arrivo alla configurazione γ'.

Mentre indicheremo con

$$\gamma \underline{\to}_n \gamma'$$

se con n passi di calcolo si passa dalla configurazione γ alla configurazione γ'. Quella definita è una relazione dove $\gamma R \gamma'$ se $\gamma \underset{\longrightarrow}{n} \gamma'$

- $\gamma R \gamma$

  infatti $\gamma \underset{\longrightarrow}{0} \gamma$

- $\gamma R \gamma'$ e $\gamma' R \gamma''$ allora $\gamma R \gamma''$

  infatti se $\gamma \underset{\longrightarrow}{n} \gamma'$ e $\gamma' \underset{\longrightarrow}{m} \gamma''$ allora $\gamma \underset{\longrightarrow}{n+m} \gamma''$

Più sinteticamente indichiamo

$$\gamma \xrightarrow[M]{*} \gamma'$$

dove $*$ vuol dire che non ci interessa più il numero dei passi di calcolo da fare ed M sta ad indicare la macchina di Turing che stiamo utilizzando.

Osservazione:
La relazione $\xrightarrow[M]{*}$ è la chiusura transitiva e riflessiva della relazione $\rightarrow$

In modo più esplicito una computazione è una successione di passi

$$\gamma_0 \rightarrow \gamma_1 \quad , \ \gamma_1 \rightarrow \gamma_2 \quad , \ \cdots \quad , \quad \gamma_n \rightarrow \gamma_{n+1}$$

che indicheremo in modo più breve con

$$\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n+1}$$

oppure con il formalismo introdotto dalla chiusura transitiva e riflessiva possiamo ancora più in breve indicarla con

$$\gamma_0 \xrightarrow[M]{*} \gamma_{n+1}$$

Daltronde spesso preferiremo scrivere tutto come

$$(q_0, \underline{\Delta} w) \xrightarrow[M]{*} (q, w')$$

Computazioni finite, infinite o in stallo

Diremo che una computazione termina con successo se dopo un numero finito di passi di calcolo lo stato diventa h, diremo che una computazione termina senza successo se si giunge ad una situazione di stallo, ovvero la funzione di transizione non è definita per alcuni valori di Σ, il che equivale a dire che il nostro impiegato legge un'istruzione che non sa eseguire. In tutti gli altri casi diremo che la computazione non termina.
Consideriamo ad esempio la seguente macchina di Turing

q	σ	$\delta(q,\sigma)$
q_0	Δ	(q_0, Δ, R)
q_0	a	$(q_0, a, -)$
q_0	#	$(q_0, a, -)$

Facendo partire la computazione dallo stato

$$(q_0, \underline{\Delta}aa\#)$$

otteniamo che questa non termina, infatti

$$(q_0, \underline{\Delta}aa\#) \to (q_0, \Delta\underline{a}a\#) \to (q_0, \Delta\underline{a}a\#) \to (q_0, \Delta\underline{a}a\#) \to \dots$$

la computazione continua per sempre restando sempre sulla stessa configurazione.

Mentre una situazione di stallo potrebbe essere

$$(q_0, \underline{\Delta}ba) \to ((q_0, \Delta\underline{b}a))$$

Non essendo definita in b la funzione di transizione allora la computazione va in stallo e quindi l'impiegato si ferma non sapendo più cosa fare.

Definiamo

$$M(w) = (q_0, \underline{\Delta}w)$$

Diremo che $M(w)$ converge se partendo dallo stato $(q_0, \underline{\Delta}w)$ la computazione termina e lo indicheremo con $M(w) \downarrow$

Diremo che $M(w)$ diverge se partendo dallo stato $(q_0, \underline{\Delta}w)$ la computazione non termina e lo indicheremo con $M(w) \uparrow$

Calcolo di funzioni

E' possibile costruire macchine di Turing in grado di calcolare funzioni, mostriamo come costruire una macchina in grado di sommare due numeri.
Per semplificare il tutto rappresentiamo i numeri con delle I ovvero il numero 1 sarà I, il numero 2 sarà II e così via il numero n sarà $III\dots I$.
Quindi noi vogliamo una macchina che faccia

$$(q_0, \underline{\Delta}II + III\#) \xrightarrow[M]{*} (h, \Delta IIIII\underline{\#})$$

Tale macchina sarà definita nel seguente modo

q	σ	$\delta(q,\sigma)$
q_0	Δ	(q_0,Δ,R)
q_0	I	(q_0,I,R)
q_0	$+$	(q_1,I,R)
q_1	I	(q_1,I,R)
q_1	$\#$	$(q_2,\#,L)$
q_2	I	$(h,\#,-)$

Facciamo quindi tutta la computazione

$$(q_0,\underline{\Delta}II{+}III\#) \to (q_0,\Delta\underline{I}I{+}III\#) \to (q_0,\Delta I\underline{I}{+}III\#) \to (q_0,\Delta II\underline{+}III\#) \to (q_1,\Delta III\underline{I}II\#) \to$$

$$\to (q_1,\Delta IIIII\underline{I}\#) \to (q_1,\Delta IIIIII\underline{\#}) \to (q_1,\Delta IIIIII\underline{\#}\#) \to (q_2,\Delta IIIIII\underline{\#}\#) \to (h,\Delta IIIII\#\#)$$

Sottointendiamo gli spazi bianchi di fine stringa perchè poniamo che gli elementi siano tutti bianchi definitivamente; a fine esecuzione non è detto che ci troviamo su un #

I problemi che si cercano di risolvere con le macchine di Turing sono: calcolare una funzione oppure decidere l'appartenenza di un certo elemento ad un insieme.

Esercizio proposto

Scrivere una macchina di Turing che termina con successo se prende in ingresso una stringa palindroma e va in stallo altrimenti.

Funzioni Turing-calcolabili

Dati due alfabeti Σ_0 e Σ_1 tali che

$$\Delta, \# \notin \Sigma_0 \cup \Sigma_1 \subseteq \Sigma$$

Consideriamo una funzione

$$f : \Sigma_0^* \to \Sigma_1^*$$

è T-calcolabile (o Turing-calcolabile) se esiste una macchina di Turing M tale che

$$f(w) = w' \iff M(w) = w'$$

NB: $M(w) = w'$ vuol dire che esiste una computazione tale che

$$(q_0, \underline{\Delta} w) \xrightarrow[M]{*} (h, \Delta w')$$

23/03/2011

Funzioni primitive ricorsive

Abbiamo fin ora consierato le funzioni come

$$f : \Sigma_0^* \to \Sigma_1^*$$

e abbiamo detto che una funzione è T-calcolabile se esiste una macchina di Turing tale che

$$f(w) = w' \iff M(w) = w'$$

senza perdere di generalità, dato che Σ^* è numerabile possiamo considerare come $\Sigma_0^* = \Sigma_1^* = \mathbb{N}$ e quindi avremo solo funzioni

$$f : \mathbb{N} \to \mathbb{N}$$

Qundi d'ora in avanti ci limiteremo a studiare questo tipo di funzioni. Cambiamo ora totalmente approccio e consideriamo una classe di funzioni dette funzioni primitive ricorsive che indicheremo con

$$\lambda \ \text{(variabili)}. \ \text{(funzione delle variabili)}$$

Ad esempio una funzione primitiva ricorsiva è

$$\lambda \ x, y. \ x + y$$

Osserviamo che se modifichiamo l'argomento

$$\lambda \ x. \ x + y$$

y diventa un parametro e non una variabile.

Chiameremo "posti" il numero di variabili di una funzione.

Definiamo C la classe delle funzioni primitive ricorsive come la minima classe che riesco a definire mediante queste istanze:

1 Esistenza della funzione nulla

$\lambda\ x_1, x_2, \ldots, x_n.0$

2 Successore

$\lambda x.x + 1$

3 Proiezione

$\lambda x_1, \ldots, x_n.x_i$

4 Composizione

Se $g_1, \ldots, g_k \in C$ sono funzioni a n posti ed $f \in C$ è una funzione a k posti, allora posso definire una nuova funzione a n posti $h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), g_2(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$ con $h \in C$

5 Ricorsione primitiva

sia $g \in C$ una funzione a $n-1$ posti e sia $h \in C$ a $n+1$ posti, allora possiamo definire una nuova funzione $f \in C$ definita dalla relazione seguente

$$\begin{cases} f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n) \\ f(x_1 + 1, x_2, \ldots, x_n) = h(x_1, f(x_1, \ldots, x_n), x_2, \ldots, x_n) \end{cases}$$

Esempio
Siano date le funzioni

$$\begin{cases} f_1(x) = x \\ f_2(x) = x + 1 \\ f_3(x_1, x_2, x_3) = x_2 \\ f_4(x_1, x_2, x_3) = f_2(f_3(x_1, x_2, x_3)) \\ f_5(0, x_2) = f_1(x_2) \\ f_5(x_1 + 1, x_2) = f_4(x_1, f_5(x_1, x_2), x_2) \end{cases}$$

non è difficile verificare che queste sono funzioni primitive ricorsive, nella seguente scrittura abbiamo per semplicità contratto la notazione, ad esempio quella completa sarebbe

$$f_1 = \lambda\ x.x_1$$

E così per le altre, mentre abbiamo scritto in sequenza f_5 che sarebbe

$$f_5(x_1, x_2) = \begin{cases} f_1(x_2) & \text{se} \quad x_1 = 0 \\ f_4(x_1 - 1, f_5(x_1, x_2), x_2) & \text{se} \quad x_1 \neq 0 \end{cases}$$

Per comodità tralasciamo questo modo di scrivere le cose e usiamo il primo esposto.
Calcoliamo $f_3(2, 3)$ allora

$$f_3(2, 3) = f_4(1, f_5(1, 3), 3)$$

A questo punto dobbiamo scegliere se sviluppare f_4 oppure f_5, il che è equivalente per il risultato ma seguire una strada piuttosto che un'altra potrebbe farci risparmiare tempo. Possiamo stabilire delle regole che sono dette regole di valutazione, ovvero si sceglie se sviluppare ad esempio sempre le parentesi più interne oppure le parentesi più esterne.
Stabiliamo che svolgiamo sempre la più interna a sinistra e quindi

$$f_3(2,3) = f_4(1, f_5(1,3), 3) = f_4(1, f_4(0, f_5(0,3)), 3) = f_4(1, f_4(0, f_1(3), 3), 3) =$$
$$= f_4(1, f_4(0,3,3), 3) = f_4(1, f_2(f_3(0,3,3)), 3) = f_4(1, f_2(3), 3) = f_4(1,4,3) =$$
$$= f_2(f_3(1,4,3)) = f_2(4) = 5$$

Osserviamo che equivalentemente $f_3(x,y) = x+y$ o come preferiremo scrivere la funzione equivalente è

$$\begin{cases} 0 + y = y \\ (x+1) + y = (x+y) + 1 \end{cases}$$

Numerabilità delle macchine di Turing

L'insieme delle macchine di Turing è numerabile, ricordiamo la definizione

$$M = (Q, \Sigma, \delta, q_0)$$

e si è già osservato che $\Sigma \simeq \mathbb{N}$, quindi senza perdere di generalità d'ora in poi supporremo che l'alfabeto si ogni macchina di Turing sia esattamente $\mathbb{N}$, in realtà c'è anche la possibilità che Σ sia finito, in tal caso è sufficiente lasciare indefinita la funzione di transizione per i valori superflui.

Facciamo lo stesso ragionamento su Q, sappiamo che questo insieme è finito (lo abbiamo imposto prima) quindi possiamo pensarlo a meno di isomorifismo come $Q \subset \mathbb{N}$, ma come prima generalizziamo e consideriamo Q uguale per tutte le macchine di Turing con $Q \simeq \mathbb{N}$.

Per quello che abbiamo detto sulle funzioni

$$\delta \subset (Q \times \Sigma) \times ((Q \cup \{h\}) \times \Sigma \times \{L, R, -\})$$

Usando il fatto che prodotto finito di insiemi numerabili è numerabile concludiamo che $\delta \simeq \mathbb{N}$

In definitiva tutto ciò ci serviva per dara una caratterizzazione delle macchine di Turing, possiamo considerare le quintuple

$$(q_i, \sigma_j, \delta_k, q_t) \qquad i,j,k,t \in \mathbb{N}$$

Queste definiscono completamente una macchina di Turing che sarà

$$M = \{(\, q_i, \sigma_j, \delta_k, q_t) \;\text{ tale che }\; (i,j,k,t) \in A \quad \text{di cardinalità finita}\}$$

Consideriamo l'applicazione

$$(Q, \Sigma, \delta, Q) \longrightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

$$(q_i, \sigma_j, \delta_k, q_t) \longrightarrow (i,j,k,t)$$

Per i discorsi fatti prima questo è un isomorifismo. Inoltre sappiamo al solito che prodotto finito di insiemi numerabili è numerabile quindi possiamo caratterizzare una macchina di Turing come una successione finita di numeri naturali, ovvero

$$M = \{\alpha_n \;\text{ tale che }\; n \in I \quad \text{di cardinalità finita } m\}$$

Quindi la successione finita $\{\alpha_n\}_{n\in I}$ caratterizza la macchina di Turing, cioè ad ogni α_i posso, per i discorsi fatti prima, associare una configurazione e viceversa.

Per concludere con la tesi ci basta mostrare che l'insieme costituito dalle successioni di numeri naturali finite è numerabile.
Sia tale insieme

$$E = \left\{\{\alpha_n\}_{n\in I} \quad \text{I di cardinalità finita } m\right\}$$

Dimostriamo questo fatto, dobbiamo associare ad ogni successione un numero naturale, sia

$$\{\alpha_n\}_{n\in I} = \{\alpha_{i_1}, \alpha_{i_2}, \dots \alpha_{i_m}\}$$

Supponiamo di ordinare i numeri primi (ad esempio in ordine crescente) $p_1, p_2, \dots,$ allora consideriamo l'applicazione

$$E \longrightarrow \mathbb{N}$$

$$\{\alpha_n\}_{n\in I} \to p_{i_1}^{\alpha_{i_1}} p_{i_2}^{\alpha_{i_2}} \dots p_{i_m}^{\alpha_{i_m}}$$

Questa applicazione è iniettiva per il teorema di fattorizzazione unica, inoltre è anche surgettiva, infatti dato un numero naturale n allora sempre per il teorema di fattorizzazione unica avremo

$$n = p_{i_1}^{\alpha_{i_1}} p_{i_2}^{\alpha_{i_2}} \dots p_{i_m}^{\alpha_{i_m}}$$

invertiamo quindi l'applicazione

$$\mathbb{N} \longrightarrow E$$

$$n = p_{i_1}^{\alpha_{i_1}} p_{i_2}^{\alpha_{i_2}} \dots p_{i_m}^{\alpha_{i_m}} \to \{\alpha_n\}$$

quindi abbiamo esibito una bigezione e per quanto detto prima, questo conclude il teorema.

Osservazione
Le funzioni primitive ricorsive terminano. La dimostrazione si può fare per induzione osservando le istanze che devono esser verificate.

Funzione di Ackermann

Osserviamo che dall'esempio precedente che abbiamo fatto di funzione primitiva ricorsiva abbiamo che la somma è una generalizzazione dell'operazione di successore.

Non è difficile vedere che il prodotto è una generalizzazione della somma

$$\begin{cases} 0 \cdot y = 0 \\ (x+1) \cdot y = (x \cdot y) + y \end{cases}$$

E andando avanti così troviamo che l'esponenziale è una generalizzazione del prodotto

$$\begin{cases} y^0 = 1 \\ y^{x+1} = y^x \cdot y \end{cases}$$

Possiamo definire una generalizzazione dell'esponenziale?

Definiamo la funzione di Ackermann nel seguente modo

$$\begin{cases} A(0,0,y) = y \\ A(0,x+1,y) = A(0,x,y) + 1 \\ A(1,0,y) = 0 \\ A(z+2,0,y) = 1 \\ A(z+1,x+1,y) = A(z, A(z+1,x,y), y) \end{cases}$$

Osserviamo che

- $A(0,x,y) = x + y$
- $A(1,x,y) = x \cdot y$
- $A(2,x,y) = y^x$
- $A(3,x,y) = y^{y^{y^{\cdots}}}$ fatto x volte.

Si può dimostrare in generale che la funzione di Ackermann non è primitiva ricorsiva, l'idea di questa dimostrazione è che la funzione di Ackermann cresce più in fretta di ogni funzione primitiva ricorsiva dato che richiama se stessa troppe volte, daltronde questa dimostrazione è molto tecnica e la tralasciamo.

Numerabilità delle funzioni primitive ricorsive

Si può dimostrare che le funzioni primitive ricorsive sono numerabili, la dimostrazione è molto simile a quella della numerabilità delle macchine di Turing, usa esattamente la stessa idea. Infatti si parte dal fatto che tutte le funzioni primitive ricorsive terminano, quindi hanno una successione finita di stati, il che possiamo immaginarlo come la successione finita di configurazioni che ha la macchina di Turing e ripetere esattamente la stessa dimostrazione.

Non tutte le funzioni sono primitive ricorsive

Ci sono funzioni che non rispettano le istanze che non appartengono alla classe C delle funzioni primitive ricorsive, un esempio è stato già mostrato prima, la funzione di Ackermann, daltronde non abbiamo dato dimostrazione di questo fatto e comunque si potrebbe pensare che questo sia l'unico caso di funzione non primitiva ricorsiva.
Quindi dimostriamo in generale questo fatto. Abbiamo già visto che le funzioni primitive ricorsive sono numerabili, dunque posso associare un numero ad ogni funzione e metterle in fila

$$f_0, f_1, \ldots, f_n, \ldots$$

Posso dunque considerare la funzione seguente

$$g(n) = f_n(n) + 1$$

Questa è calcolabile ma non è primitiva ricorsiva, infatti non può comparire nella lista che ho fatto, infatti se per assurdo $g = f_j$ allora

$$g(j) = f_j(j) = f_j(j) + 1$$

Dato che non è possibile che un numero naturale sia uguale ad un suo successore si ha l'assurdo.

μ-ricorsione

Aggiungiamo ancora un'altra istanza per definire la classe delle funzioni primitive generali dette μ-ricorsive

6 μ-ricorsione

se $g(y, x_1, \ldots, x_n)$ è una funzione primitiva ricorsiva allora la funzione

$$\phi(x_1, \ldots, x_n) = \mu\ y.g(y, x_1, \ldots, x_n) = 0$$

è μ-ricorsiva.

Se y non esiste la funzione non è definita se invece ci sono più $y_i \in I$ che soddisfano $g(y_i, x_1, \ldots, x_n) = 0$, allora prendiamo

$$\phi(x_1, \ldots, x_n) = \min_I y_i$$

Ad esempio se $g(y, x) = y + 1$ allora $\phi(3) \uparrow$

Quindi abbiamo le funzioni T-calcolabili e le funzioni μ-calcolabili, si riesce a mostrare che questi due concetti sono equivalenti e che una funzione è T-calcolabile se e solo se è μ-calcolabile.

Tesi di Church-Turing

Le funzioni intuitivamente calcolabili (mediante algoritmi) sono tutte e sole le T-calcolabili.

Differenza tra algoritmi e funzioni

Sia $\phi \in \mathbb{N} \times \mathbb{N}$ una funzione, un algoritmo è un modo per calcolarla. Dato che ci sono tanti algoritmi per calcolare una funzione (ad esempio basta svolgere in ordine diverso le operazioni) ci aspettiamo che gli algoritmi siano molti di più delle funzioni, vedremo questa cosa formalmente nel Padding lemma.

Notazione

Supponendo di aver numerato tutte le macchine di Turing come mostrato prima allora data una funzione $\phi \subset \mathbb{N} \times \mathbb{N}$, se la macchina M_n calcola la funzione allora chiamerò ϕ_n la funzione calcolata da M_n.

Numerabilità delle funzioni calcolabili

Le funzioni calcolabili sono $\mathbb{N}$

dimostrazione
Come conseguenza della tesi di Church-Turing abbiamo che una funzione è calcolabile se e soltato se è T-calcolabile, se

$$f(w) = w' \iff M(w) = w'$$

Dove quando scriviamo

$$M(w) = w'$$

intendiamo che presa la configurazione w con una computazione finita la macchina di Turing giunge alla configurazione (terminante) w', sinteticamente $w \xrightarrow[*]{w} {}'$. Pertanto ad ogni funzione associamo almeno una macchina di Turing e le macchine di Turing sono numerabili. Quindi le funzioni calcolabili sono numerabili.

Funzioni non calcolabili

Ci sono funzioni non calcolabili

dimostrazione
Abbiamo mostrato prima che usando la tesi di Church-Turing le funzioni calcolabili sono numerabili. Sia dunque

$$F = \{f_i \quad \text{tale che} \quad i \in \mathbb{N}\}$$

l'insieme delle funzioni calcolabili. Consideriamo dunque la funzione $g(n) = f_n(n) + 1$, dunque se g fosse calcolabile allora $g = f_j$ per qualche $j \in \mathbb{N}$ ma allora

$$g(j) = f_j(j) = f_j(j) + 1$$

ed è assurdo che un numero naturale sia uguale al suo successore. Ciò prova la tesi.

Osservazione:
Si è usata la stessa argomentazione che si è usata per dimostrare che esistono funzioni che non sono primitive ricorsive, questa argomentazione è nota come processo diagonale e si usa spesso per questo tipo di dimostrazioni.

Padding lemma

Ogni funzione calcolabile ϕ ha un numero numerabile di indici ϕ_n nel senso che si è dato prima.

Cosa vuol dire:

- Sia ϕ una funzione calcolabile, per la tesi di Church-Turing allora questa funzione è T-calcolabile
- Una funzione è T-calcolabile se esiste una macchina di Turing che la calcola, quindi esiste una M macchina di turing tale che $\phi(w) = w' \iff M(w) = w'$, ovvero con una computazione terminante la macchina di Turing M partendo dalla configurazione w giunge alla configurazione w' (che sinteticamente indichiamo con $w \xrightarrow{*} w'$)

- Il punto è che questa macchina di Turing non è unica ma ne esistono una quantità numerabili. Quindi per ogni funzione ϕ esistono $\mathbb{N}$ macchine di Turing che la calcolano nel senso specificato prima.

Dimostrazione
Supponiamo che la funzione ϕ sia calcolata dalla macchina di $M = (Q, \Sigma, \delta, q_0)$. Posso creare una nuova macchina di Turing M' semplicemente ingrandendo l'insieme degli stati, ad esempio prendo $q' \notin Q$ e quindi considero $Q' = Q \cup \{q'\}$, chiaramente dovrò anche modificare la funzione di transizione e mi basterà definire

$$\delta(q', \sigma) = (h, \#, R) \quad \forall \sigma \in \Sigma$$

In questo modo la macchina M' continua a calcolare ϕ. Ma posso continuare a fare questa costruzione con $q'' \notin Q'$ e cotruire M'', quindi posso costruire una quantità almeno numerabile di macchine di Turing che calcolano la funzione ϕ. Dato che le macchine di Turing sono numerabili allora il numero di macchine di Turing che calcola ϕ è numerabile. Ciò prova la tesi.

Esempio
Possiamo vederla sotto l'aspetto dei linguaggi di programmazione: se ho scritto un programma che esegue un certo algoritmo posso scriverne una quantità numerabile che lo eseguono, infatti mi basta aggingere al programma una riga vuota oppure posso aggingere una variabile a e fare l'assegnazione $a = 0$, il risultato del programma non cambia e questa procedura posso farla quante volte voglio.

24/03/2011

Codifiche

La volta scorsa si è usato il termine codifica, vediamo cosa vuol dire senza dare una definizione troppo formale. Codificare qualcosa alla buona vuol dire assegnargli un numero in modo coerente con quello che si sta facendo. Ad esempio abbiamo visto che è possibile codificare le macchine di Turing, ovvero è possibile assegnare ad ogni macchina di Turing un unico numero naturale e viceversa. Al tempo stesso abbiamo visto che è possibile codificare una funzione primitiva ricorsiva dato che possiamo mettere in bigezione anche questo insieme con $\mathbb{N}$. Possiamo inoltre codificare le computazioni, ovvero possiamo associare ad ogni computazione un numero in modo univoco (e viceversa). Non lo dimostriamo formalmente anche perchè la dimostrazione è uguale a quella fatta sulle macchine di Turing ma mostriamo come farla seguendo la stessa linea.

Si è più volte detto che una computazione terminante è una successione finita di configurazioni che in modo sintetico indichiamo con

$$\gamma_1 \xrightarrow{*} \gamma_k$$

mentre in modo esteso

$$\gamma_1 \to \gamma_2 \to \cdots \to \gamma_k$$

abbiamo già osservato come è possibile associare un numero ad ogni configurazione, di conseguenza una computazione sarà daterminata da una successione finita di numeri

$$\{\ \alpha_i\}_{i \in I} \quad \text{tale che} \quad I \subset \mathbb{N} \quad \text{di cardinalità finita}$$

E abbiamo già mostrato come associare ad ogni successione finita di numeri un numero naturale (è proprio la dimostrazione fatta sulle macchine di Turing).

D'ora in avanti useremo la parola codifica per intendere proprio ciò, non ci importa sapere quale codifica stiamo usando ma solo che esista una codifica, ovvero una bigezione con l'insieme dei numeri naturali. Quindi quando parleremo di codifica di una computazione o di una macchina di Turing o di una funzione ci riferiremo proprio a queste costruzioni.

Teorema (di forma normale) di Kleene

Esiste un predicato $T(i,x,y)$ ed una funzione $U : \mathbb{N} \to \mathbb{N}$ entrambi ricorsivi primitivi tale che

$$\phi_i(x) = U(\lambda\ y.T(i,x,y))$$

dove quest'ultima scritta in forma più estesa sarebbe

$$\phi_i(x) = U(\bar{y})$$

dove

$$\bar{y} = \min\{y \quad \text{tale che} \quad T(i,x,y) \quad \text{è vero}\}$$

E dove scegliamo il predicato come

$$T(i,x,y) = \text{vero} \iff y \text{ è una codifica di una computazione terminante di } \phi_i(x)$$

T è noto come predicato di Kleene.

Dimostrazione
Se y è una codifica di una computazione terminante di $\phi(x)$ vuol dire che ad y possiamo associare una computazione

$$\phi_i(x) \xrightarrow{*} \gamma$$

La funzione U possiamo sceglierla come la funzione di decodifica, cioè data la codifica di una computazione U restituisce il risultato della computazione. In tal modo è ovvio che

$$\phi_i(x) = U(\lambda\ y.T(i,x,y))$$

Teorema di enumerazione

Esiste z tale che per ogni i, x vale che

$$\phi_i(x) = \phi_z(i,x)$$

Dimostrazione
Uso il teorema di Kleene e quindi so che

$$\phi_i(x) = U(\lambda y.T(i,x,y) = 1)$$

quindi U è una funzione calcolabile, $\lambda y.T(i,x,y) = 1$ è una funzione calcolabile (è μ-ricorsiva) allora la loro composizione è calcolabile, dunque per la tesi di Church-Turing avrà una codifica, la chiamo z, dunque $U(\lambda y.T(i,x,y) = 1) = \phi_z(i,x)$. Andado a sostituire

$$\phi_i(x) = \phi_z(i,x)$$

che è appunto la tesi.

NB: la y non compare come variabile nella ϕ_z perchè si trova per tentativi.

Macchina di Turing universale

Il teorema di enumerazione ci dice che esiste una "macchina di Turing universale", con ciò intendiamo una macchina di Turing in grado di simulare ogni macchina di Turing. La costruzione di questa è complicata ma se ne può dare un'idea. Innanzitutto non si perde di generalità se si considera che l'impiegato possa leggere contemporaneamente tre nastri (ovvero le strisce con le istruzioni) e seguire conemporaneamente tre istruzioni, l'unica cosa che cambirà sarà la funzione di transizione che sarà definita come

$$\delta(q,(\sigma_1,\sigma_2,\sigma_3)) = (q',(\sigma_1',\sigma_2',\sigma_3'),(d_1,d_2,d_3))$$

dove $d_i \in \{ L, R, - \}$ è la direzione i-esima.

Non è difficile convincersi che questa continua ad essere una macchina di Turing per come l'abbiamo costruita. A questo punto possiamo simulare la macchina M con i dati in ingresso x nel seguente modo

Δ	$\rho(M)$	x	...
Δ	$\tau(x)$	...	
Δ	q_0	...	

dove $\rho(M)$ è la codifica della macchina di Turing, $\tau(x)$ la codifica di x.
Inoltre alcuni stati possono esser anche dei simboli, quindi non è detto che $Q \cap \Sigma = \emptyset$, daltonde per comodità assegnamo ad ogni stato un elemento dell'alfabeto (un simbolo) attraverso una codifica, in questo modo gli stati possono esser letti.
Non è difficile capire che questa è una macchina di Turing universale ma questa non ha la pretesa di essere una costruzione rigorosa ma vole solo dare l'idea di come sia possibile costruirla.

Teorema del parametro (o S-M-N)

Esiste S calcolabile, totale e iniettiva tale che

$$\forall x, y, i \qquad \lambda\ y\ .\phi_i(x, y) = \phi_{S(i,x)}(y)$$

Significato:

Quello che dice il teorema del parametro è che dato un programma di due variabili esiste un algoritmo ricorsivo che fornisce in uscita un programma di 1 variabile che fornisce gli stessi risultati del primo e che codifica l'altra variabile. In realtà questa è la versione S-1-1 del teorema, ne esiste una più generale ma esula dai nostri obbiettivi.

dimostrazione:
Costruisco la funzione S: prendo i, lo decodifico per ottenere una macchina di Turing M_i e le do in ingresso x. Quello appena descritto è a tutti gli effetti un algoritmo che prende come ingresso (i, x), per la tesi di Church-Turing allora posso associarli una macchina di Turing M_h dove sarà $h = S(i, x)$.

versione generale:

Informalmente, il teorema generale dice che, dato un programma che prende in entrata $m + n$ variabili, esiste un algoritmo ricorsivo che fornisce in uscita un

programma di n variabili che fornisce gli stessi risultati del primo e che codifica le altre m variabili.

Teorema di espressività

Ogni formalismo universale ha il teorema di enumerazione o il teorema s-m-n

Significato:

Un formalismo universale è ad esempio quello delle funzioni primitive ricorsive oppure quello di Turing, ad ogni modo tutti i formalismi universali hanno il teorema di enumerazione o il teorema s-m-n. (Non diamo una definizione formale di formalismo universale).

Teorema del punto fisso (secondo di Kleene)

Per ogni funzione f calcolabile e totale esiste un numero n tale che

$$\phi_n = \phi_{f(n)}$$

dimostrazione
definisco la seguente funzione

$$\psi(u,z) = \begin{cases} \phi_{\phi_u(u)}(u) & \text{se} \quad \phi_u(u) \downarrow \\ \text{indefinita altrimenti} \end{cases}$$

Questa funzione è calcolabile, quindi per la tesi di Church-Turing ha un indice, allora uso il teorema del parametro

$$\phi_{S(i,u)}(z) = \phi_i(u,z) = \psi(u,z)$$

definisco

$$d(u) = \lambda\ u\ .S(i,u)$$

quindi

$$\phi_{S(i,u)}(z) = \phi_{d(u)}(z)$$

Ma allora considero $f \circ d$ questa è una funzione calcolabile, quindi per la tesi di Church-Turing $f(d(u))$ è calcolata da una macchina

$$\phi_v(x) = f(d(x))$$

allora

$$\phi_{d(v)} = \phi_{\phi_v(v)} \quad \text{dato che} \quad \phi_v(v) \downarrow \quad \forall v$$

sia

$$d(v) = n$$

dico che n è il punto fisso, infatti

$$\phi_n = \phi_{d(v)} = \phi_{\phi_v(v)} = \phi_{f(d(v))} = \phi_{f(n)}$$

ciò prova la tesi.

28/03/2011

Ci poniamo il problema di stabilire se una dato elemento appartiene ad un insieme.

Funzione caratteristica

Definiamo la funzione caratteristica di un insieme I nel seguente modo

$$\chi_I(x) = \begin{cases} 1 & \text{se} \quad x \in I \\ 0 & \text{se} \quad x \notin I \end{cases}$$

Insiemi ricorsivi

Diremo che un insieme I è ricorsivo se la sua funzione caratteristica χ_I è calcolabile e totale.
Indichiamo la classe degli insiemi ricorsivamente numerabili con $\mathcal{E}$

Esempio
Consideriamo la seguente funzione

$$g(x) = \begin{cases} 1 & \text{se la congettura di Goldbach è vera} \\ 0 & \text{altrimenti} \end{cases}$$

Questa è una funzione calcolabile dato che è costante (non sappiamo se vale sempre 1 o sempre 0 ma ad ogni modo le funzioni costanti sono calcolabili), quindi esiste una macchina di Turing che la calcola.

Esempio
Consideriamo il seguente insieme

$$\{(i, x, h) \ \text{ tale che } \ \phi_i(x) \downarrow \quad \text{in meno di } k \text{ passi}\}$$

Questo insieme è ricorsivo infatti la sua funzione caratteristica è calcolabile, per stabilire se una tripletta (i, x, h) appartiene a questo insieme ci basta prendere l'i-esima macchina di Turing, dargli come ingresso x e fargli fare h passi e vedere se si è giunti ad una configurazione terminante.
Questo algoritmo è intuitivamente calcolabile, quindi per la tesi di Church-Turing concludiamo che la funzione caratteristica è calcolabile quindi l'insieme è induttivo.

Insiemi ricorsivamente numerabili

Diremo che un insieme I è ricorsivamente numerabile se esiste un indice i tale che

$$I = dom(\phi_i) = \{n \ \text{ tale che } \ \phi_i(n) \downarrow\}$$

Ovvero un insieme è ricorsivamente numerabile se esiste una funzione calcolabile parziale (ovvero non tolale) che ha per dominio esattamente l'insieme I.
Denoteremo con $\mathcal{RE}$ la classe di tali insiemi.
Spesso chiameremo ϕ funzione quasi-catatteristica dell'insieme I.

Proposizione:
Se I è ricorsivo allora è ricorsivamente numerabile, quindi $\mathcal{E} \subseteq \mathcal{RE}$.

dimostrazione
La funzione quasi-caratteristica di I è la sua funzione caratteristica (se supponiamo $0 \notin \mathbb{N}$ abbiamo terminato altrimenti con qualche passaggio in più si ha lo stesso risultato).

Proposizione:
Se I è ricorsivamente enumerabile e anche il suo complementare $\bar{I}$ lo è allora I e $\bar{I}$ sono ricorsivi.

dimostrazione
Mostriamo prima un approccio sbagliato che si può avere a questo tipo di problema.
Dato che I è ricosivamente enumerabile possiamo descriverlo per qualche i fissato come

$$I = \{n \quad \text{tale che} \quad \phi_i(n) \downarrow\}$$

Quindi verrebbe da dire che la funzione caratteristica di I è la seguente

$$\chi_I(x) = \begin{cases} 1 & \text{se} \quad \phi_i(x) \downarrow \\ 0 & \text{altrimenti} \end{cases}$$

Ma questa funzione non è calcolabile!
Scrivere la dimostrazione formale di questo fatto è poco costruttivo dato che è solo tecnica, daltronde l'idea della dimostrazione è la seguente:
Sappiamo che alche $\bar{I}$ è un insieme ricorsivamente numerabile, quindi anche lui sarà per qualche j della forma

$$\bar{I} = \{n \quad \text{tale che} \quad \psi_i(n) \downarrow\}$$

con ψ funzione calcolabile. Allora possiamo considerare una macchina di Turing che fa fare un passo di calcolo per trovare $\phi_i(x)$ e dopo un passo per trovare $\psi_i(x)$, poi ancora un altro passo per $\phi_i(x)$ e via così in modo alternato. Dato che le possibilità sono 2, o x appartiene all'insieme oppure non ci appartiene, questa computazione è terminante, quindi se la computazione termina con ϕ_i allora $x \in I$ altrimenti se termina in ψ_i allora $x \notin I$. Pertanto questo è un algoritmo intuitivamente calcolabile, quindi per la tesi di Church-Turing possiamo associarli una macchina di Turing e quindi una funzione calcobile. Da qui è semplicissimo costruire la funzione indicatrice dell'insieme I.

Caratterizzazione alternativa degli insiemi ricorsivamente numerabili

Diremo che un insieme I è ricorsivamente numerabile se è vuoto oppure se è l'immagine di una funzione calcolabile totale.

Notazione

Denotiamo con $rec = \{\phi \quad \text{tale che} \quad dom(\phi) = \mathbb{N}\}$

Mostriamo ora che le caratterizzazioni date degli insieme ricorsivamente numerabili sono equivalenti.

Supponiamo di avere un insieme ricorsivamente numerabile nel senso della prima definizione, ovvero abbiamo la funzione quasi-caratteristica ϕ dell'insieme I, mostriamo come costruire una f totale che abbia come immagine esattamente l'insieme I.

Prima di dimostrare il tutto osserviamo che le difficoltà legate a questa dimostrazione sono nel fatto che chiediamo che la funzione f sia totale, ovvero con dominio $\mathbb{N}$.

La dimostrazione si basa sul tassellamento di $\mathbb{N} \times \mathbb{N}$ ovvero la numerazione di questo insieme. Consideriamo dunque la seguente tabella

	0	1	2	3	4	...
0	0	2	5	9	14	...
1	1	4	8	13	...	
2	3	7	12	...		
3	6	11	...			
4	10	...				
...	...					

Quindi la tabella l'abbiamo costruita seguento il tassellamento di $\mathbb{N} \times \mathbb{N}$, quindi leggiamo in diagonale a zig-zag seguendo la numerazione crescente.
Se costruiamo una tabella con la stessa logica di questa ca con il seguente significato: nella cella (n, m) ci metteremo $\phi_i(n)$ con m passi di calcolo (i è un indice fissato dalla definizione di insieme ricorsivamente numerabile).

numero passi \ argomento	0	1	2	3	4	...
0	$0.\phi_i(0)$	$2.\phi_i(0)$	$5.\phi_i(0)$	$9.\phi_i(0)$	$14.\phi_i(0)$	...
1	$1.\phi_i(1)$	$4.\phi_i(1)$	$8.\phi_i(1)$	$13.\phi_i(1)$	...	
2	$3.\phi_i(1)$	$7.\phi_i(1)$	$12.\phi_i(1)$	...		
3	$6.\phi_i(1)$	$11.\phi_i(1)$	...			
4	$10.\phi_i(1)$	...				
...	...					

Dove poniamo che $m.\phi_i(n)$ è la codifica della configurazione della i-esima macchina di Turing dopo m passi di calcolo.

Riassumendo

- nella cella $(0,0)$ ci sarà $\phi_i(0)$ dove l'i-esima macchina di Turing ha fatto 0 passi di calcolo
- nella cella $(1,0)$ ci sarà $\phi_i(0)$ dove l'i-esima macchina di Turing ha fatto 1 passi di calcolo
- nella cella $(0,1)$ ci sarà $\phi_i(1)$ dove l'i-esima macchina di Turing ha fatto 0 passi di calcolo
- ...
- nella cella (n,m) ci sarà $\phi_i(n)$ dove l'i-esima macchina di Turing ha fatto m passi di calcolo

Per ipotesi abbiamo che $dom(\phi) \neq \emptyset$ quindi se continuiamo a scrivere questa lista arriviamo con un numero finito di passi $\bar{n}$ ad una configurazione terminante.

Siamo ora in grado di costruire la funzione f la cui immagine sarà il nostro insieme I e la costruiremo con ragionamenti simili a quelli fatti fin ora.
Consideriamo al solito la numerazione di $\mathbb{N} \times \mathbb{N}$ e costruiamo al solito la tabella

	0	1	2	3	4	...
0	0	2	5	9	14	...
1	1	4	8	13	...	
2	3	7	12	...		
3	6	11	...			
4	10	...				
...	...					

Per ogni $k \in \mathbb{N}$ ci associamo due numeri (n,m) che sono le coordinate nella tabella che abbiamo discegnato, quindi poniamo

$$f(k) = \begin{cases} m & \text{se } \phi_i(k) \text{ converge in } n \text{ passi} \\ \bar{n} & \text{altrimenti} \end{cases}$$

Dove $\bar{n}$ lo abbiamo definito prima e $\phi_i(k)$ converge in n passi vuol dire che l'i-esima macchina di Turing giunge ad una configurazione terminante dopo n passi

di calcolo. Chiaramente f è una funzione totale e la sua immagine è esattamente l'insieme I. Cio prova che la prima definizione è equivalente alla seconda.
In viceversa è immediato.

Osservazione
In generale vale che $\mathcal{RE} \subset \mathcal{R}$ e l'inclusione è stretta, ovvero ci sono insiemi che sono ricorsivamente numerabili ma non ricorsivi, ovvero ammettono solo una funzione quasi-caratteristica.

Mostriamo un esempio

$$K = \{x \quad \text{tale che} \quad \phi(x) \downarrow\}$$

Questo insieme è ricorsivamente numerabile, ad esempio lo si può vedere come conseguenza del teorema di enumerazione, daltronde non è ricorsivo, infatti non ammette una funzione caratteristica calcolabile.
Per dimostrarlo supponiamo per assurdo che una tale funzione χ_K esista e sia calcolabile, definiamo allora la funzione

$$f(x) = \begin{cases} \phi_k(x) + 1 & \text{se} \quad \chi_K(x) = 1 \\ 0 & \text{altrimenti} \end{cases}$$

Quindi se χ_K fosse calcolabile allora lo sarebbe anche f, daltronde per Church-Turing se f fosse calcobile avrebbe un indice i, quindi si avrebbe

$$f = \phi_i$$

allora valutando la funzione in i

$$f(i) = \phi_i(i) = \begin{cases} \phi_i(i) + 1 & \text{se} \quad \phi_i(i) \quad \text{converge} \\ 0 & \text{se} \quad \phi_i(x) \quad \text{non converge} \end{cases}$$

In ogni caso abbiamo l'assurdo, nel primo caso un numero è uguale al suo successore, nel secondo abbiamo che il fatto che $\phi_i(i)$ non converge implica che $\phi_i(i)$ converge.

Aggingiamo che per quanto detto prima abbiamo che

$$\bar{K} = \{x \quad \text{tale che} \quad \phi_x(x) \uparrow\}$$

Non è ricorsivo (quindi a maggir ragione non è ricorsivamente numerabile). Diremo che $\bar{K}$ è un insieme indecidibile.

Definiamo inoltre $co - \mathcal{RE}$ gli insiemi che hanno il complementare ricorsivamente numerabile.

Abbiamo la seguente catena dove tutte le inclusioni sono strette

$$\{\text{insiemi finiti}\} \subset \mathcal{R} \subset \mathcal{RE} \subset (co - \mathcal{RE} \cup \mathcal{RE})$$

Proposizione
Unione numerabile di insiemi ricorsivi non è un insieme ricorsivo

dimostrazione
Basta esibire un controesempio, prendiamo

$$A_h = \{(x, x, h) \quad \text{tale che} \quad \phi_x(x) \quad \text{termina in meno di } h \text{ passi}\}$$

Gli insiemi A_k sono tutti ricorsivi ma non è difficile trovare che

$$\bigcup_{h \in \mathbb{N}} A_h = K$$

e per quanto visto prima questo non è ricorsivo.

Proposizione
Intersezione finita di insiemi ricorsivi è ricorsiva, non vale per l'intersezione numerabile

dimostrazione
Basta dimostrare la tesi con due insiemi, per induzione segue il caso finito. Se I e J sono insiemi ricorsivi allora l'indicatrice di $I \cap J$ sarà $\chi_{I \cap J}(x) = \chi_I(x)\ \chi_J(x)$. Per mostrare che non è vero nel caso numerabile basta prendere la dimostrazione della proposizione precedente e ripeterla con i complementari.

Esempio (compilatore)
Supponiamo di avere un compilatore $C_L^{L \to A}$ dove L in basso vuol dire che è scritto con in linguaggio L, in alto $L \to A$ vuol dire che prende in ingresso un programma sorgente scritto in L e in uscita da un programma oggetto (eseguibile) scritto in linguaggio A.
Supponiamo di voler ricavare un compilare $C_A^{L \to A}$ scritto in linguaggio A che prende in input un programma sorgente scritto in L e in output da un programma oggetto scritto in A. Allora basterà considerare la composizione

$$C_L^{L \to A}(C_L^{L \to A}) = C_A^{L \to A}$$

Esempio (problema della fermata)
Il seguente insieme è indecidibile

$$K_0 = \{(x, y) \quad \text{tale che} \quad \phi_x(y) \downarrow\}$$

e stabilire l'appartenenza di una coppia (x, y a questo insieme è noto come problema della fermata, diremo che questi problemi sono indecidibili.

Un'idea della dimostrazione di questo fatto è la seguente: osserviamo che $x \in K \iff (x, y) \in K_0$, quindi in un certo senso il problema dell'appartenenza a K

è equivalente a quello dell'appartenza a K_0, quindi l'idecidibilità di K implica quella di K_0, con la definizione di riduzione che segue sarà possibile completare la dimostrazione.

Riduzione

Definiamo riduzione una funzione $f : A \to B$ tale che

- f è iniettiva
- $x \in A \iff f(x) \in B$

In tal caso scriveremo $A \leq_f B$ per dire che A si riduce tramite f a B

Classe di riduzioni

Data una classe di riduzioni

$$F = f \quad \text{tale che} \quad f : A \to B \quad \text{è una riduzione}$$

Sciveremo che

$$A \leq_F B \quad \iff \quad \exists f \in F \quad \text{tale che} \quad A \leq_f B$$

E diremo che A si riduce a B tramite la classe di riduzioni F.

Classifiche

Date due classi di problemi $\mathcal{D}$ e $\mathcal{E}$ e una classe di riduzioni F, diremo che F è una classifica se

- $A \leq_F A$ (ovvero l'identità è un elemento di F)
- $f, g \in F \Rightarrow ffn \in F$ (transitività, o chiusura per composizione)
- $A \leq_F B \ , \ B \in \mathcal{D} \Rightarrow A \in \mathcal{D}$ (struttura di ideale)
- $A \leq_F \ , \ B \in \mathcal{E} \Rightarrow \mathcal{E}$

Esempio
$\leq_{rec}$ è una classifica per $\mathcal{R}$ e $\mathcal{RE}$

Problemi ardui e problemi completi

Diremo che un problema H è arduo per una classe di problemi C rispetto ad alla classe di riduzioni F se

$$\forall A \in C \quad A \leq_F H$$

Diremo che H è completo se è arduo e se $H \in C$.

Esempio
K è completo in $\mathcal{RE}$ rispetto $\leq_{rec}$

dimostrazione
Sia

$$A = \{x \quad \text{tale che} \quad \phi_i(x) \downarrow\}$$

A partire da ϕ_i posso costruire $\psi(x, y) = \phi_i(x)$, questa è una funzione calcobile, quindi ha un indice $\phi_j(x, y)$, applico il teorema del parametro e ho

$$\phi_{S(i,x)}(y) = \phi_j(x, y)$$

Quindi posso descrivere l'insieme A come

$$A = \{x \quad \text{tale che} \quad \phi_{(S(j,x))}(y) \downarrow\}$$

Dato che non ho dipendenza dal parametro y allora

$$A = \{x \quad \text{tale che} \quad \phi_{(S(j,x))}(S(j, x)) \downarrow\}$$

$$A = \{x \quad \text{tale che} \quad (S(j, x)) \in K\}$$

Devo ora eliminare la dipendenza da y, quindi definisco

$$f(x) = \lambda\, x.\ S(j, x)$$

Inserendo la f segue la tesi, pertanto K è completo in $\mathcal{RE}$ rispletto alla classe di riduzione delle funzioni totali.

04/04/2011

Esercizio

Consideriamo l'insieme

$$K = \{x \ \text{ tale che } \ \phi_x(x) \downarrow\}$$

e consideriamo anche l'insieme

$$\text{TOT} = \{x \ \text{ tale che} \forall y \quad \phi_x(y) \downarrow\}$$

Osserviamo che praticamente TOT contiene tutti e solo gli indici delle funzioni che sono totali.

Vogliamo mostrare che

$$K \leq_{rec} \text{TOT}$$

Per mostrarlo dobbiamo esibire una riduzione.
Definiamo

$$\psi(x,y) = \begin{cases} 1 \quad \text{se} \quad x \in K \\ \text{indefinita altrimenti} \end{cases}$$

Questa è una funzione calcolabile, quindi per la tesi di Church-Turing posso associargli un indice

$$\phi_i(x,y) = \psi(x,y)$$

Posso daltronde applicare il teorema del parametro e quindi

$$\phi_{S(i,x)}(y) = \phi_i(x,y)$$

Definisco la funzione

$$f(x) = \lambda x.S(i,x)$$

E quindi alla fine ho

$$\phi_{f(x)}(y) = \phi_{S(i,x)}(y)$$

Ricapitoliamo mostrando tutte le uguaglianze che abbiamo ottenuto fin ora

$$\phi_{f(x)}(y) = \phi_{S(i,x)}(y) = \phi_i(x,y) = \psi(x,y)$$

A questo punto vogliamo mostrare che f è la riduzione cercata, ovvero devo mostrare che

$$x \in K \Rightarrow x \in \text{TOT}$$

infatti

$$x \in K \Rightarrow \forall y \;\; \phi_{f(x)}(y) = 1 \Rightarrow f(x) \in \text{TOT}$$

Daltronde

$$x \notin K \Rightarrow \forall y \;\; \phi_{f(x)} \uparrow \Rightarrow f(x) \notin \text{TOT}$$

Quindi f è la riduzione richiesta.

Richiamo
Perchè la funzione che abbiamo definito all'inizio $\psi(x,y)$ è calcolabile?
Rifacciamo il ragionamento del tutto generale sulle funzioni calcolabili.

ϕ_x è la funzione ϕ calcolata dall'x-esima macchina, questo vuol dire che

$$\forall n \quad \phi_x(n) = m \iff M_x(n) \xrightarrow{*} (h, \Delta, n)$$

Ma vuol anche dire che

$$\forall n \quad \phi_x(n) \;\; \text{è indefinita} \quad \iff M_x(n) \uparrow$$

Dovrebbe ora esser chiaro perchè la funzione $\psi(x,y)$ è calcolabile.

Osservazione
TOT non può esser un insieme ricorsivo per la proprietà della struttura di ideale.

Osservazione
Se ho una proprietà in TOT ce l'ho anche in K.

Insiemi di indici che rispettano le funzioni

Diremo che un insieme di indici I rispetta le funzioni se

$$\forall x \in I \text{ vale che } \phi_x = \phi_y \Rightarrow y \in I$$

Osservazione
Se un insieme di undici che rispetta le funzioni non è vuoto allora non può avere cardinalità finita per il padding lemma.

Esempio
TOT è un insieme che rispetta le funzioni, questa anzi è una rappresentazione delle classi funzioni (come vedremo dopo).

Lemma

Sia A un insieme di indici che rispetta le funzioni con $A \neq \emptyset$ e $A \neq \mathbb{N}$, allora K si riduce ad A oppure K si riduce a $\bar{A}$, ovvero in forma stringata

$$K \leq_{rec} A \quad \text{oppure} \quad K \leq_{rec} \bar{A}$$

dimostrazione
Sia i_0 un indice tale che ϕ_{i_0} sia ovunque indefinita. Quindi succede che $i_0 \in A$ oppure $i_0 \in \bar{A}$.
Allora dato che $A \neq \emptyset$ esiste almeno un indice i_1 tale che $i_1 \in A$, ma dato che A rispetta le funzioni abbiamo $\phi_{i_0} \neq \phi_{i_1}$, definisco dunque

$$\psi(x,y) = \begin{cases} \phi_{i_1}(y) & \text{se } x \in K \\ \phi_{i_0}(y) & \text{se } x \notin K \end{cases}$$

Per i motivi detti prima questa è una funzione calcolabile, quindi facendo gli stessi passaggi fatti nell'esercizio precedente otteniamo

$$\phi_{f(x)}(y) = \psi(x,y)$$

Mostriamo ora che f è la riduzione richiesta.

$$x \in K \Rightarrow \phi_{f(x)} = \phi_{i_1} \Rightarrow f(x) \in A$$

La prima implicazione è dovuta alla definizione, la seconda è dovuta al fatto che A rispetta le funzioni. In modo speculare otteniamo

$$x \notin K \Rightarrow \phi_{f(x)} = \phi_{i_0} \Rightarrow f(x) \notin A$$

Quindi la tesi.

Teorema di Rice

Sia $\mathcal{A}$ una classe di funzioni calcolabili e sia $A = \{x \text{ tale che } \phi_x \in \mathcal{A}\}$, allora A è ricorsivo se e soltanto se $\mathcal{A} = \emptyset$ oppure $\mathcal{A}$ contiene tutte le funzioni calcolabili.

Osservazione
Per non confondersi, si ricorda che ϕ_x è la funzione calcolata da M_x

Significato
Se voglio mostrare una certa proprietà su una funzione f non è sufficiente mostrarla sulle sue rappresentazioni (questo è il discorso che si accennava su TOT)

dimostrazione
Se $A = \emptyset$ oppure $A = \mathbb{N}$ allora la tesi è ovvia, se invece non siamo in uno di questi due casi, per il lemma precedente abbiamo la tesi dato che A si riduce a K oppure $\bar{A}$ si riduce a K ed entrambi non sono insiemi ricorsivi.

Osservazione
$K \leq_{rec}$ TOT ma non è facile da dimostrare. Anche i seguenti insiemi non sono ricorsivi:

- FIN $= \{x \text{ tale che } dom(\phi_x) \text{ è finito}\}$
- INF $= \mathbb{N} \setminus$ FIN
- REC $= \{x \text{ tale che } dom(\phi_x) \text{ è ricorsivo}\}$
- CONST $= \{x \text{ tale che } \phi_x \text{ è costante}\}$
- EXT $= \{x \text{ tale che } \phi_x \text{ è estendibile a funzione totale}\}$

Osservazione
Ci sono funzioni che non si estendono a funzioni totali, ad esempio

$$\phi(x) = \begin{cases} 1 \text{ se } x \in K \\ \text{indefinita altrimenti} \end{cases}$$

Non si estende a funzione totale, per vederlo basta provare a supporre per assurdo che si estenda e usare la tesi di Church-Turing e dare un indice a tale funzione, non è difficile giungere ad un assurdo.

05/04/2011

Monoidi generati da alfabeti

Dati due insiemi A e B definiamo l'operatore di concatenazione nel seguente modo formale

$$A \times B \to A \cdot B$$

$$(a, b) \to a \cdot b$$

Dove $a \cdot b$ è un elemento formale, in generale non chiediamo che $\cdot$ rispetti particolari regole, chiediamo che rispetti l'associatività

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

quindi in generale indicheremo in modo più stringato con

$$a \cdot b \cdot c$$

Quindi abbiamo l'insieme

$$A \cdot B = \{a \cdot b \ \text{ tale che } \ a \in A \ \ b \in B\}$$

Facciamo ora questa operazione su un alfabeto Σ definendo induttivamente

$$\begin{cases} \Sigma^0 = \{\epsilon\} \\ \Sigma^{n+1} = \Sigma \cdot \Sigma^n \end{cases}$$

Dove ϵ è l'elemento neutro, detto anche stringa vuota, ed è un elemento che ha la seguente proprietà

$$a \cdot \epsilon = \epsilon \cdot a = a \qquad \forall a \in \Sigma$$

Quindi definiamo l'insieme di tutte le stringhe

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$$

Quindi Σ^* è un monoide con l'identità ϵ e il prodotto di concatenazione è associativo ma non commutativo,

$$w \cdot (v \cdot u) = (w \cdot v) \cdot u$$

Se

$$x = wvu$$

diremo che w è il prefisso di x, v è una sottostringa di x e u è il suffisso di x.

Definiamo la lunghezza di una stringa

$$|w| = n \iff w \in \Sigma^n$$

Definiamo l'insieme delle stringhe non vuote

$$\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$$

Definiamo i linguaggi

$$L \subseteq \Sigma^*$$

Chiaramente possiamo definire con il solito modo la concatenazione di linguaggi, l'unione, ecc
Come possiamo definire un linguaggio?
Abbiamo già visto che è possibile definire un linguaggio e stabilire se una certa stringa ci appartiene usando le macchine di Turing, vogliamo ora definire i linguaggi in modo diverso.

Grammatiche generative (o a struttura di fase)

Una grammatica generativa è una quadrupla

$$G = (N, \Sigma, P, S)$$

Dove abbiamo

- N è un alfabeto detto dei non terminali
- Σ è una alfabeto detto dei terminali
- P è una relazione, ovvero $P \subseteq (N \cup \{\epsilon\})^+(N \cup \{\Sigma\})^*$, a volte indiceremo la relazione con $\rightarrow$
- $S \in N$ è detto simbolo di partenza

In genere indicheremo gli elementi di N con lettere maiuscole $N = \{A, B, \dots\}$ mentre gli elementi di Σ con lettere minuscole $\Sigma = \{a, b, \dots\}$

Diremo che

$$\gamma\alpha\delta \Rightarrow_G \gamma\beta\delta \quad \text{se} \quad \alpha \rightarrow \beta \quad \text{ovvero se } \alpha \text{ è in relazione con } \beta$$

Considero la chiusura transitiva e riflessiva di $\Rightarrow_G$ e la chiamo $\Rightarrow^*$, quindi in forma più stringata indicherò $w \Rightarrow^* w'$.

La costruzione che stiamo facendo è equivalente a quella fatta per i passi di calcolo.
Quindi data una grammatica G possiamo definire il linguaggio generato da tale grammatica

$$L(G) = \{w \in \Sigma^* \text{ tale che } S \Rightarrow w\}$$

Osservazione:
Per definire una grammatica mi basta dire quali sono le relazioni (ovvero dare l'insieme P).

Grammatiche di tipo zero (o generali)

Diremo che una grammatica è di tipo zero se è data da relazioni del tipo

$$\alpha \to \beta \quad \text{con} \quad \alpha \in (N \cup \Sigma)^* \quad , \quad \beta \in (N \cup \Sigma)^*$$

Quindi non pongo alcuna condizione, pertanto dirò che $L(G)$ è un linguaggio di tipo zero, osserviamo che $L(G)$ è di tipo ricorsivamente enumerabile.

Esempio

- $S \to aAb$
- $aA \to aaAb$
- $A \to \epsilon$

Ad esempio facciamo una derivazione

$$S \Rightarrow aAb \Rightarrow aaAbb \cdots \Rightarrow a^n Ab^n \quad n \geq 1$$

Quindi

$$L(G) = \{a^n b^n \quad \text{tale che} \quad n \geq 1\}$$

Grammatiche di tipo uno (o contestuali)

Diremo che una grammatica è di tipo uno se è data da relazioni del tipo

$$\gamma A \delta \to \gamma \beta \delta \quad \text{con} \quad A \in N \quad \gamma, \delta, \beta \in (N \cup \Sigma)^+$$

Esempio

- $S \to aAb|ab$
- $aA \to aaAb|aab$

Dove | va inteso come "oppure", ovvero da S possiamo passare ad aAb oppure ad ab

Grammatiche di tipo due (libere da contesto)

Diremo che una grammatica è di tipo due se è data da relazioni del tipo

$$A \to \beta \qquad \text{con} \qquad A \in N \quad \beta \in (N \cup \Sigma)^+$$

Esempio

$S \to aSb|ab$

Questo genera un linguaggio dove ogni parola contiene tante a quante b.

Esempio(parentesi bilanciate)

- S $\to (S)|SS|()$
Questo genera un linguaggio in cui il numero delle parentesi aperte è uguale al numero delle parentesi chiuse e ogni sottostringa sinistra contiene più parentesi aperte che chiuse. Questo fatto si dimostra in modo semplicissimo per induzione.

Osservazione (inclusione delle grammatiche)
Se una grammatica è di tipo 2 allora è anche di tipo 1, se una grammatica è di tipo 1 allora è anche di tipo zero

$$\text{tipo } 2 \Rightarrow \text{tipo } 1 \Rightarrow \text{tipo } 0$$

Questa inclusione vale anche per i linguaggi ed è detta gerarchia di Chomsky

Grammatiche di tipo tre (lineari destre o sinistre)

Diremo che una grammatica è di tipo due se è data da relazioni del tipo

$$A \to aB|b \qquad \text{con} \qquad A, B \in N \quad a, b \in \Sigma$$

Esempio (linguaggio non libero)

- $S \to aSBc|abc$
- $cB \to Bc$
- $bB \to bb$

Si può mostrare che

$$L(G) = \{\ a^n b^n c^n \ \text{ tale che } \ n \geq 1\}$$

non è libero

Espressioni

D'ora in avati per le espressioni denoteremo con E il simbolo di partenza

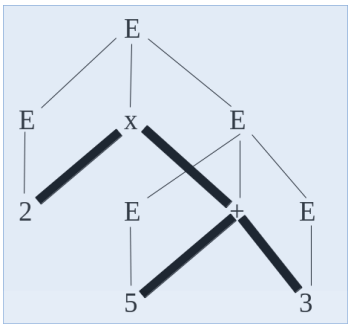
$$E \to E + E | E \times E | 2 | 3 | 5$$

Dove al solito | va letta come oppure, allora partendo da E possiamo avere due derivazioni sinistre differenti

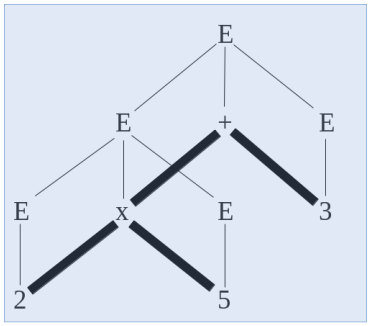
$$E \Rightarrow E \times E \Rightarrow 2 \times E \Rightarrow 2 \times E + E \Rightarrow 2 \times 5 + E \Rightarrow 2 \times 5 + 3$$

$$E \Rightarrow E + E \Rightarrow E \times E + E \Rightarrow 2 \times E + E \Rightarrow 2 \times 5 + E \Rightarrow 2 \times 5 + 3$$

Daltronde pur arrivando alla stessa conclusione se costruiamo l'albero di derivazione (inutile definirlo formalmente si capisce cos'è con questo esempio) avremo nel primo caso



nel secondo caso

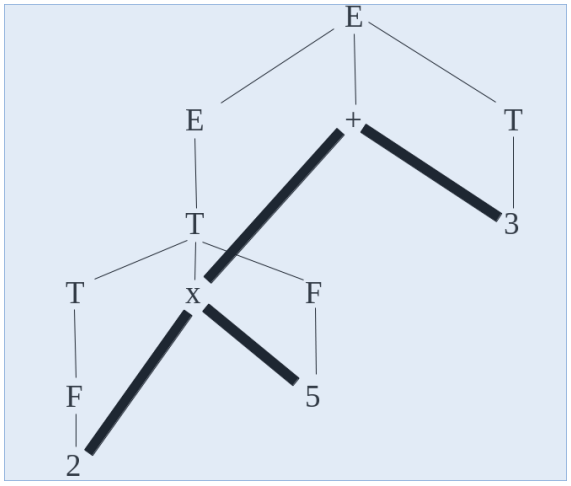


Quelli in figura sono detti alberi di derivazione, se invece vediamo solo la parte marcata allora quelli li chiameremo alberi di sintassi astratta. Si vede che nel primo caso il risultato è 16 mentre nel secondo è 30 (le operazioni si fanno seguendo una visita anticipata a sinistra, quindi partendo dalle foglie), pertanto diremo che questa grammatica è ambigua dato abbiamo ottenuto $2 \times 5 + 3$ in due modi ma il risultato è diverso. Quindi diremo che questa grammatica è ambigua avendo due derivazioni canoniche sinistre diverse.

Esempio(grammatica non ambigua)
Correggiamo la grammatica precedente per renderla non ambigua, sostanzialmente cerceremo di far scendere il $\times$ nell'albero di sintassi astratta in modo che quando risolviamo l'espressione si eseguono prima i prodotti e poi le somme.

- $E \rightarrow E + T|T$
- $T \rightarrow T \times F|F$
- $F \rightarrow 2|3|5$

Ad esempio dal caso precedente ora abbiamo



07/04/2011

Linguaggi regolari e automi a stati finiti

Ricordiamo che un linguaggio è regolare se è definito da una grammatica del seguente tipo

$$A \rightarrow aB|b$$

I linguaggi regolari sono usati per l'analisi lessicale.

Mostriamo ora un altro approccio per definire un linguaggio regolare, ricodiamo la definizione di automa, questo si definisce come una quintupla

$$(Q, \Sigma, \delta, q_0, F)$$

- Q è l'insieme degli stati con cardinalità finita
- Σ è l'insieme dei simboli, detto anche alfabeto
- $\delta : Q \times \Sigma \rightarrow Q$ è la funzione di transizione
- $F \subseteq Q$ è l'insieme degli stati finali

Possiamo estendere la funzione di transizione nel seguente modo

$$\delta^* : Q \times \Sigma^* \rightarrow Q$$

definita come

$$\begin{cases} \delta^*(q, \epsilon) = q \\ \delta^*(q, aw) = \delta^*(\delta(q, a), w) \end{cases}$$

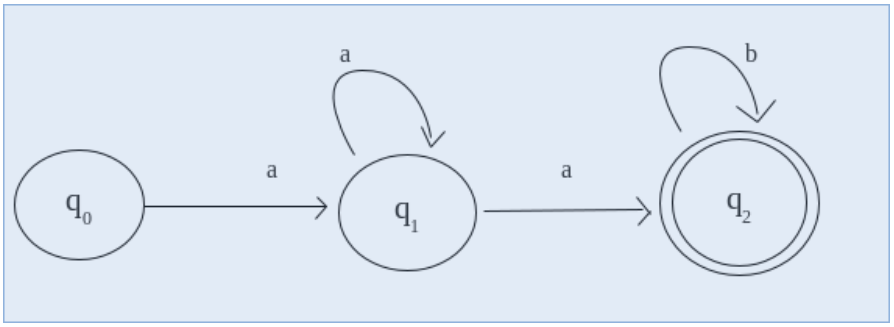
In generale possiamo descrivere uun linguaggio regolare con un automa a stati finiti e diremo che il linguaggio che genera è

$$L_A = \{w \quad \text{tale che} \quad \delta^*(q_0, w) \in F \quad \text{con} \quad q_o \in Q\}$$

Qindi in generale è possibile associare ad ogni linguaggio regolare un automa a stati finiti e viceversa ad ogni automa a stati finiti possiamo associare un linguaggio regolare.

Osservazione
Gli automi che fin ora abbiamo considerato sono deterministici, quelli non deterministici hanno una sola differenza, la funzione di transizione di definisce in modo diverso $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ Dove $\mathcal{P}(Q)$ sono le parti di Q, ovvero ad ogni coppia (simbolo,stato) associamo più stati. Non ci soffermiamo su questo aspetto dato che si può dimostrare che i concetti sono equivalenti. Ovvero ogni automa deterministico è anche non deterministico e viceversa se un automa è non deterministico ne esiste uno equivalente deterministico che accetta lo stesso tipo di linguaggio.

Esempio
Consideriamo il seguente automa



Questo è un automa deterministico e il linguaggio accettato è

$$L_A = \{a^n b^m \quad \text{tale che} \quad m, n > 0\}$$

Relazioni invarianti destre

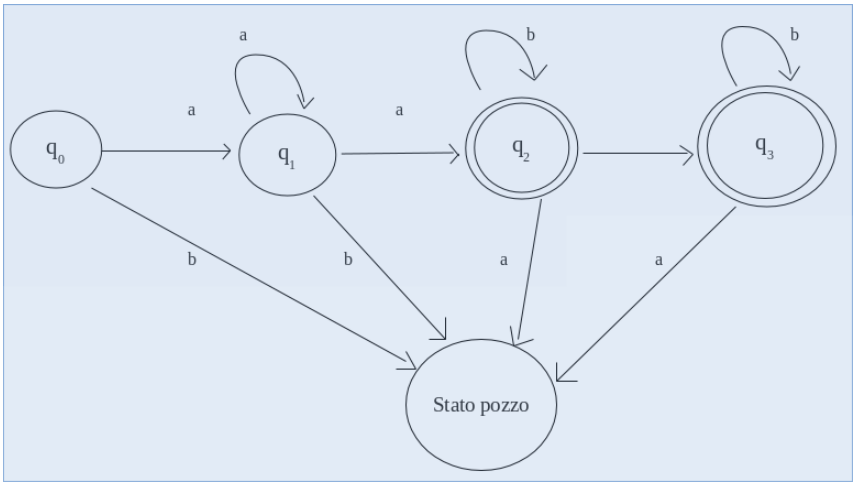
Data una certa relazione $\equiv$ su un certo insieme diremo che questa è invariante destra se

$$x \sim y \Rightarrow xz \sim yz \quad \forall z$$

Quindi nel nostro caso se al posto di x e y ci mettiamo delle stringhe abbiamo che la relazione è invariante destra.

- q_0 contiene la classe $[\epsilon]$
- q_1 contiene la classe $[a^n]$
- q_2 contiene la classe $[a^n b^m]$

Esempio(automa equivalente)
Il seguente automa è equivalente al precedente



Definizione di un linguaggio regolare

In definitiva possiamo descrivere un linguaggio regolare in tre modi

- Con un a grammatica, quindi $L(G) = \{w \text{ tale che } s \Rightarrow^* w\}$
- Con un automa a stati finiti, quindi $L_A = \{w \quad \text{tale che} \quad \delta^*(q_0, w) \in F \quad \text{con} \quad q_o \in Q\}$
- Con una relazione destra, quindi $L = \bigcup_{\text{finita}} \{\text{classi di } \sim\}$

Osservazione
La classe dei lingiaggi regolari è chiusa per intersezione, complemento, unione, concatenazione ecc.

Pumping lemma (per un linguaggio regolare)

Se L è un linguaggio regolare, allora esiste un numero n che dipende solo dal linguaggio tale che per ogni $w \in L$ con $|w| \geq n$ esistono $x, y, z \in \Sigma^*$ con le seguenti proprietà

- $w = xyz$
- $y \neq \epsilon$
- $|xy| \leq n$

- $\forall k \qquad xy^k z \in L$

Conclusione
L'analisi lessicale quindi la si fa con le grammatiche regolari, vediamo ora come opera l'analizzatore sintattico, cioè come dare l'analisi sintattica, quindi parlaremo di lingiaggi liberi e di automi a pila.

Linguaggi liberi e automi a pila non deterministici

Consideriamo ora i linguaggi liberi, ad esempio troviamo che

$$S \to aSb|ab$$

Definisce un linguaggio

$$L = \{a^n b^n | n > 0\}$$

Questo è libero ma non regolare (ad esempio perchè non vale il pumping lemma), o consideriamo il linguaggio delle parentesi bilanciate

$$S \to (S)|SS|()$$

Vedremo che tali linguaggi si descrivono con automi a pila.

Definiamo un automa a pila come

$$P = (Q, \Sigma, \gamma, \Delta, q_0, z)$$

- Q è l'insieme degli stati
- Σ è l'insieme dei simboli
- γ è l'insieme dei simboli che metto nella pila
  (notazione: indicheremo gli elementi come $\gamma = \{z, u, v, \dots\}$)
- Δ è la relazione di transizione con $\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \gamma) \times (Q \times \gamma^*)$

definiamo un passo come

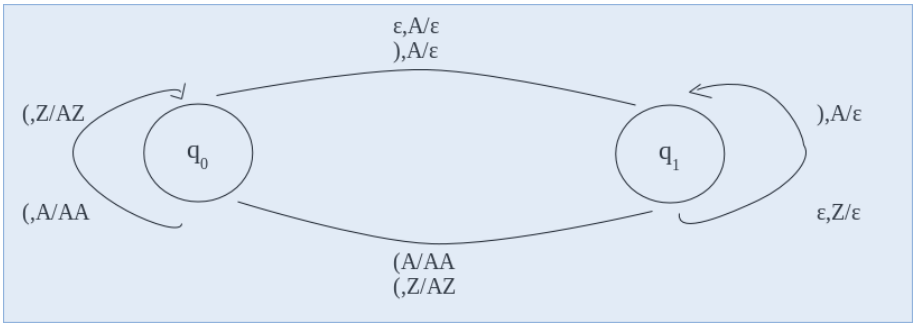
$$(q, w, U) \Rightarrow (q', w', ') \iff (q', ') \in \Delta(q, a, U) \text{ e } \begin{cases} w = aw' & \text{se } a \in \Sigma \\ w = w' & \text{se } a = \epsilon \end{cases}$$

Il linguaggio accettato dall'automa P è

$$L_P = \{w \in \Sigma^* \text{ tale che } (q_0, w, z) \Rightarrow^* (q, \epsilon, \epsilon)$$

Quindi osserviamo subito che l'automa P non ha stati termianali ma l'accettazione di una stringa avviene per pila vuota.

Esempio



Questo automa a pila accetta il linguaggio delle parentesi bilanciate, il simbolo / sta ad indicare che devo sostituire il carattere a sinistra del simbolo / con quello a destra.
Ad esempio se vogliamo vedere se (()()) è nel linguaggio, facendo i passaggi

$$< q_0, Z, (()()) > \Rightarrow < q_0, AZ, ()()) > \Rightarrow < q_1, AZ, ()) > \Rightarrow < q_0, AAZ,)) >$$

$$\Rightarrow < q_1, AZ,) > \Rightarrow < q_1, Z, \epsilon > \Rightarrow (q, \epsilon, \epsilon)$$

Nota
Abbiamo detto che questo automa a pila è non deterministico, daltronde si possono definire automi a pila deterministici ma questa volta non abbiamo il risultato che avevamo prima, con gli automi non deterministici si descrivono linguaggi che non si descrivono con automi deterministici, quindi i due concetti non sono equivalenti.

Pumping lemma (per un linguaggio libero)

Con una formulazione simile al pumping lemma come enunciato nel caso dei linguaggi regolari, abbiamo che esiste un numero n dipendente solo dal linguaggio L tale che se $|z| > n$ allora esistono $u, v, w, x, y \in \Sigma^*$ tali che

- $z = uvwxy$
- $vx \neq \epsilon$
- $\forall k \quad uv^k wx^k y \in L$

07/04/2011

Grammatiche monotone

Diremo che una grammatica è monotona se le sue produzioni sono del tipo

$$\alpha \to \beta \qquad \text{tale che} \qquad |\alpha| \leq |\beta|$$

Si può dimostrare che per ogni grammatica monotona G esiste una grammatica contestuale G' tale che $L_G = L_{G'}$.

Vogliamo ora occuparti della semantica, quindi introdurremo tre approcci:

- approccio assiomatico
- approccio denotazionale
- approccio operazionale

Cenni sulla semantica operazionale

L'approccio della operazionale consiste nel associare ad ogni coppia (programma,dati) dei dati, ovvero

$$(P \times \text{DATI}) \rightarrow \text{DATI}$$

E quindi modificando i dati. Questo approccio sarà visto meglio dopo.

Cenni sulla semantica assiomatica

L'approccio assiomatico è caratterizzato da una tripletta

$$(\mathcal{P}, P, Q)$$

- $\mathcal{P}$ è l'insieme delle precondizioni
- P è il programma
- Q è l'insieme delle postcondizioni (o risultato)

Esempio
Prendiamo come insieme di precondizioni

$$\mathcal{P} = \{x \geq 0, y > x\}$$

E supponiamo che il programma P sia costituito da una successione di istruzioni separate dal punto e virgola. Quindi ad esempio poniamo

$$P \qquad x := x + 1 \ ; \ y := y - x$$

Quindi se eseguiamo il programma trasformiamo l'insieme delle precondizioni nell'insieme delle postcondizioni, quindi

$$\{x \geq 0, y > x\} \ \ x := x + 1 \Rightarrow \{x > 0, y \geq x\}$$

Quindi a sinistra abbiamo precondizioni e programma, a destra dopo $\Rightarrow$ abbiamo le postcondizioni relative al programma (in questo caso il programma è solo la prima istruzione). Andando avanti con la seconda istruzione

$$\{x > 0, y \geq x\}\, y := y - x \Rightarrow \{x > 0, y \geq 0\}$$

Quidni il risultato dell'esecuzione del programma P con le preconzioni $\mathcal{P}$ è l'insieme $Q = \{x > 0, y \geq 0\}$ Quindi possiamo scrivere in forma stringata

$$\{x \geq 0, y > x\} \;\; x := x + 1, y := y - x \Rightarrow \{x > 0, y \geq 0\}$$

Come regola generale, se il programma P_0 con le precondizioni $\mathcal{P}$ da come postcondizioni Q_0 e se il programma P_1 con le precondizioni Q_0 da come postcondizioni Q, allora possiamo scrivere il seguente

$$\frac{\mathcal{P}P_0Q_0 \wedge Q_0P_1Q}{\mathcal{P}P_0; P_1Q}$$

Ovvero $\wedge$ si legge come "e", quindi quello che facciamo è esattamente quello che si è fatto nell'esempio, spezziamo il programma in due pezzi ed eseguiamo prima la prima metà con le precondizioni date e poi la seconda metà con le postcondizioni che avviamo avuto dall'esecuzione precedente.
Chiaramente questo discorso ha una generalizzazione ovvia e possiamo dividere un programma in quanti pezzi vogliamo e vedere passo per passo come si trasforma l'insieme precondizioni-postcondizioni.

Possiamo inoltre includere il costrutto if-then-else come

$$\frac{\mathcal{P}P_0Q_0 \wedge Q_0P_1Q}{\mathcal{P} \;\; \text{if (condizione) then} \;\; P_0 \;\; \text{else} \;\; P_1; Q}$$

Cioè date le precondizioni $\mathcal{P}$ possiamo porre una condizione affinchè si esegua P_0 oppure P_1, ad esempio se $\mathcal{P}$ è un certo insieme di precondizioni che coinvolgolo la variabile x e ad esempio abbiamo il programma $P_0 \quad x := \sqrt{x}$ e il programma $P_1 \quad x := \sqrt{-x}$ allora dato che l'estrazione di radice è definita solo su numeri positivi è sensato che la condizione sia

$$\mathcal{P} \;\; \text{if} \;\; x \geq 0 \;\; \text{then} \;\; P_0 \;\; \text{else} \;\; P_1$$

Semantica denotazionale

Mostriamo ora un altro approccio, quello denotazionale.
Lo avevamo già introdotto alle prime lezioni e consiste nell'associare ad ogni programma sorgente un programma oggetto (eseguibile), quindi associare ad ogni programma una funzione da DATI in DATI

$$P \rightarrow (\text{DATI} \rightarrow \text{DATI})$$

Pensiamo al programma come definito prima, ovvero una successione di istruzioni separate dal punto e virgola e riprendiamo lo stesso esempio di prima per capire come funziona la semantica computazionale.
Per far ciò sarà necessario introdurre il concetto di configurazione per un programma, evitiamo di dare una definizione formale ma alla buona questo sarà una coppia: le istruzioni del programma che devono essere ancora eseguite e lo stato

della memoria, ad esempio riprendendo l'esempio di prima la configurazione iniziale sarà

$$< x := x + 1; y := y - x, \begin{array}{|c|c|}\hline \text{x} & 3 \\ \hline \text{y} & 5 \\ \hline \end{array} >$$

Associamo ad ogni programma una funzione detta significato, ad esempio in questo caso il significato sarà

$$[\![x := x + 1; y := y - x]\!]$$

Dove per applicato ad un insieme di dati indicheremo

$$[\![x := x + 1; y := y - x]\!] \begin{array}{|c|c|}\hline \text{x} & 3 \\ \hline \text{y} & 5 \\ \hline \end{array}$$

Osserviamo che

$$[\![x := x + 1; y := y - x]\!] = [\![y := y - x]\!][\![x := x + 1]\!]$$

In generale se C_1 e C_2 sono dei programmi allora

$$[\![C_1; C_2]\!] = [\![C_2]\!][\![C_1]\!]$$

Quindi assoceremo ad ogni programma $P \quad C_1, \ldots, C_n$ una funzione detta significato $[\![C_1; \ldots; C_n]\!]$ Riprendiamo l'esempio

$$< x := x + 1; y := y - x, \begin{array}{|c|c|}\hline \text{x} & 3 \\ \hline \text{y} & 5 \\ \hline \end{array} > \rightarrow < y := y - x, \begin{array}{|c|c|}\hline \text{x} & 4 \\ \hline \text{y} & 5 \\ \hline \end{array} > \rightarrow \begin{array}{|c|c|}\hline \text{x} & 4 \\ \hline \text{y} & 1 \\ \hline \end{array}$$

Semantica su espressioni aritmetiche

Sia $n \in \mathbb{Z}$ e $x \in$ Variabili se E è una espressione allora definiamo

$$E ::= n | x | E_1 + E_2 | E_1 - E_2 | E_1 \times E_2$$

Definiamo la funzione di interpretazione semantica $\mathcal{A}$ nel seguente modo

$$\mathcal{A} : \text{Espressioni} \rightarrow (\text{Store} \rightarrow \mathbb{Z})$$

Dove per Store si intende la memoria, ad esempio nel caso precedente Store era la tabella

x	4
y	1

Quindi la funzione di interpretazione della semantica associa ad ogni espressione una funzione che va da Store a $\mathbb{Z}$.

Definiamo più formalmente

$$\text{Store} = \{\sigma : \text{Variabili} \rightarrow \mathbb{Z} \quad \text{tale che } \sigma \text{ sia calcolabile}\}$$

Quindi Store è la classe di funzioni che associa ad una variabile un numero intero.

Per la funzione di interpretazione della semantica facciamo due richieste

- $\mathcal{A}$ ⟦n ⟧= id (identità)
  Quindi la funzione di interpretazione della semantica ha come punti fissi i numeri (cosa abbastanza sensata)
- $\mathcal{A}$ ⟦x ⟧$_\sigma = \sigma(x)$
  Equivalentemente lo scriveremo come $\mathcal{A}$ ⟦x ⟧$_\sigma = \lambda\ \sigma.\sigma(x)$
- $\mathcal{A}[\![E_1 + E_2]\!] = \mathcal{A}[\![E_1]\!] + \mathcal{A}[\![E_2]\!]$
  Dove a sinistra + è solo un simbolo definito sulle espressioni, a destra + ha il significato di operazione tra interi e ad esempio è il vero +. Chiaramente questa regola deve valere anche per il prodotto × ed altre operazioni. Quindi si deve esser in grado di associare ad ogni operazione tra espressioni un'operazione tra interi e deve esser rispettara una sorta di regola di omomorfismo.

Esempio
Sia x una variabile e $\sigma(x) = 2$, allora

$$\mathcal{A}[\![3 + x \times 5]\!]_\sigma = \mathcal{A}[\![3]\!]_\sigma + \mathcal{A}[\![x \times 5]\!]_\sigma = \mathcal{A}[\![3]\!]_\sigma + \mathcal{A}[\![x]\!]_\sigma \times \mathcal{A}[\![5]\!]_\sigma = 3 + 2 \times 5 = 13$$

Principio di composizionalità

Chiederemo che il significatp di un oggetto composto sia determinato dall'operazione di composizione e dai significati dei componenti.

Ad esempio supponiamo di estendere

$$E ::= n|x|E_1 + E_2|E_1 - E_2|E_1 \times E_2| - E_1$$

Allora la definizione

$$\mathcal{A}[\![E_1]\!] = \mathcal{A}[\![0 - E_1]\!]$$

Non va bene perchè non rispetta il principio di composizionalità.
La definizione corretta è

$$\mathcal{A}[\![- E_1]\!] = 0 - \mathcal{A}[\![E_1]\!]$$

11/04/2011

Ricapitolazione

Ricordiamo che abbiamo definito le espressioni come

$$E ::= n|x|E_1 + E_2$$

Questa è una definizione ricorsiva, n è un numero e x una variabile, questi sono detti casi base, con $E_1 + E_2$ facciamo una ricorsione, osserviamo che $+$ è una generica operazione (per indendersi può essere un prodotto, un elevamento a potenza, ecc), la formulazione più corretta sarebbe stata

$$E ::= n|x|E_1 \text{ operazione } E_2$$

ma è inutile appensatire la notazione.
Inoltre avevamo definito σ : Variabili $\to \mathcal{Z}$, dove abbiamo poi definito la semantica

$$\mathcal{A} : \text{Espressioni} \to (\text{Store} \to \mathbb{Z})$$

e abbiamo richiesto che rispetti

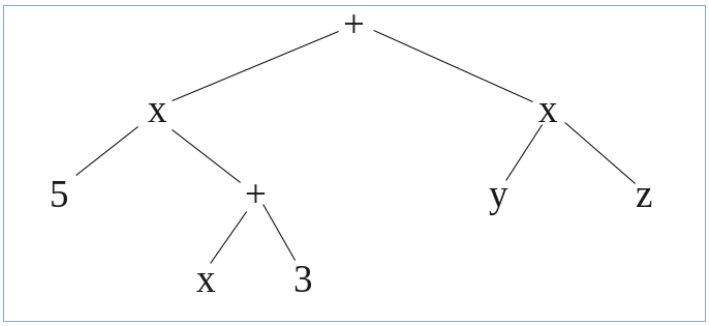
- $\mathcal{A}[\![n]\!] = \lambda\sigma.n$
- $\mathcal{A}[\![x]\!] = \lambda\sigma.\sigma(x)$
- $\mathcal{A}[\![E_1 + E_2]\!] = \mathcal{A}[\![E_1]\!]$ più $\mathcal{A}[\![E_2]\!]$

Anche qui per "più" si intende l'operazione tra interi su $\mathbb{Z}$, chiaramente sarebbe come prima più corretto scriverci "operazione tra interi" ma non appesantiamo troppo la notazione e sciviamo solo "più" o a volte direttamente "+", quindi sottointendiamo una sorta di struttura di omomorfismo.

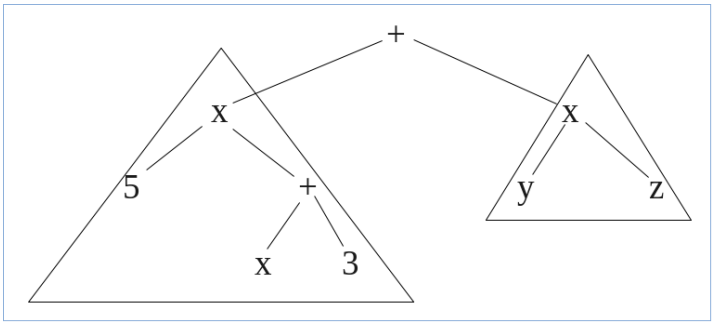
Definiamo $Var(E)$ come l'insieme delle variabili che appaiono in E Teniamo a mente che una espressione altro non è anche un albero.
Se ad esempio abbiamo l'espressione

$$5 \times (x + 3) + (y \times x)$$

gli associamo l'albero



allora $Var(E)$ saranno le variabili che compaiono nell'espressione, subito si vede che sono x, y, z daltronde possiamo definirle induttivamente come le variabili che compaiono nel sottoalbero destro unite a quelle che appaiono nel sottoalbero sinistro



Quindi induttivamente possiamo trovare l'insieme delle variabili che compaiono in una espressione E, formalmente

- $Var(n) = \emptyset$
- $Var(x) = \{x\}$
- $Var(E_1) = I_1$
- $Var(E_2) = I_2$
- $Var(E_1 + E_2) = I_1 \cup I_2$

Quindi data una espressione E induttivamente la svuotiamo e troviamo le variabili che compaiono.

La semantica come funzione totale e principio di induzione strutturale

Vale a seguente proprietà:

$$\forall x \in Var(E), \sigma(x) \in \mathbb{Z} \text{ allora } \exists n \text{ tale che } \mathcal{A}[\![E]\!]_\sigma = n$$

Quindi per ogni variabile contenuta in una espressione e una σ che la valuta esiste il risultato dell'espressione.

Per dimostrare questo risultato è necessario introdurre il principio di induzione strutturale.

Principio di induzione strutturale
Se una proprietà P

- Vale per i casi base n ed x
- Il fatto che valga per E_1 ed E_2 implica che vale per $E_1 + E_2$ (passo induttivo)

Allora la proprietà vale per tutte le espressioni.

Usiamo ora il principio di induzione strutturale per dimostrare la proprietà enunciata prima.
Per i casi base abbiamo

- $\mathcal{A}[\![n]\!]_\sigma = n$
- $\mathcal{A}[\![x]\!]_\sigma = \sigma(x) = n$

L'ipotesi induttiva dice che

$$\mathcal{A}[\![E_1]\!]_\sigma = m_1 \quad \mathcal{A}[\![E_2]\!]_\sigma = m_2$$

Quindi facciamo il passo induttivo

$$\mathcal{A}[\![E_1 + E_2]\!]_\sigma = \mathcal{A}[\![E_1]\!]_\sigma \text{ più } \mathcal{A}[\![E_2]\!]_\sigma = m_1 \text{ più } m_2 = m$$

Quindi

$$\mathcal{A}[\![E_1 + E_2]\!]_\sigma = m$$

E questa è proprio la proprietà che abbiamo enunciato prima.

Possiamo dimostrare anche un'altra proprietà, supponiamo di avere due memorie (store), ovvero i cassetti in cui sono memorizzate le variabili, allora se

$$\forall x \in Var(E) \text{ con } \sigma(x) = \sigma'(x) \ \Rightarrow \mathcal{A}[\![E]\!]_\sigma = \mathcal{A}[\![E]\!]_{\sigma'}$$

Per dimostrare questa proprietà bisogma procedere come prima, ovvero verifichaimo i casi base

$$\mathcal{A}[\![n]\!]_\sigma = \sigma(n)$$
$$\mathcal{A}[\![n]\!]_{\sigma'} = \sigma'(n)$$

ma per ipotesi $\sigma(n) = \sigma'(n)$ allora vale che $\mathcal{A}[\![n]\!]_\sigma = \mathcal{A}[\![n]\!]_{\sigma'}$
Stesso discorso con l'altro caso base

$$\mathcal{A}[\![x]\!]_\sigma = \sigma(x)$$

$$\mathcal{A}[\![x]\!]_\sigma = \sigma'(x)$$

Quindi anche qui concludiamo dalle ipotesi che $\mathcal{A}[\![x]\!]_\sigma = \mathcal{A}[\![x]\!]_{\sigma'}$
Facciamo ora il passo induttivo

$$\mathcal{A}[\![E_1 + E_2]\!]_\sigma = \mathcal{A}[\![E_1]\!]_\sigma \text{ più } \mathcal{A}[\![E_2]\!]_\sigma = \mathcal{A}[\![E_1]\!]_{\sigma'} \text{ più } \mathcal{A}[\![E_2]\!]_{\sigma'} = \mathcal{A}[\![E_1 + E_2]\!]_{\sigma'}$$

Semantica naturale

Definiamo un nuovo tipo di semantica di tipo operazionale detta semantica naturale

$$\mathcal{N} : \text{Espressioni } \times \text{ Store} \to \mathbb{Z}$$

Nella semantica naturale avremo l'insieme delle configurazioni

$$\Gamma = \{< E, \sigma >, n\}$$

Ovvero l'insieme delle configurazioni è fatto dalle coppie $< E, \sigma >$ e dai numeri $n \in \mathbb{Z}$, definiamo il passo come una relazione

$$\twoheadrightarrow \subseteq \Gamma \times \mathbb{Z}$$

Quindi avremo che il passo sarà del tipo

$$< E, \sigma > \twoheadrightarrow n$$

quindi definiamo la semantica come

$$\mathcal{N}[\![E]\!]_\sigma = n \iff < E, \sigma > \twoheadrightarrow n$$

dove dobbiamo definire la regola di inferenza nei casi base

- $< n, \sigma > \twoheadrightarrow n$
- $< x, \sigma > \twoheadrightarrow m \quad \text{se} \quad m = \sigma(x)$

mentre per le espressioni se vale

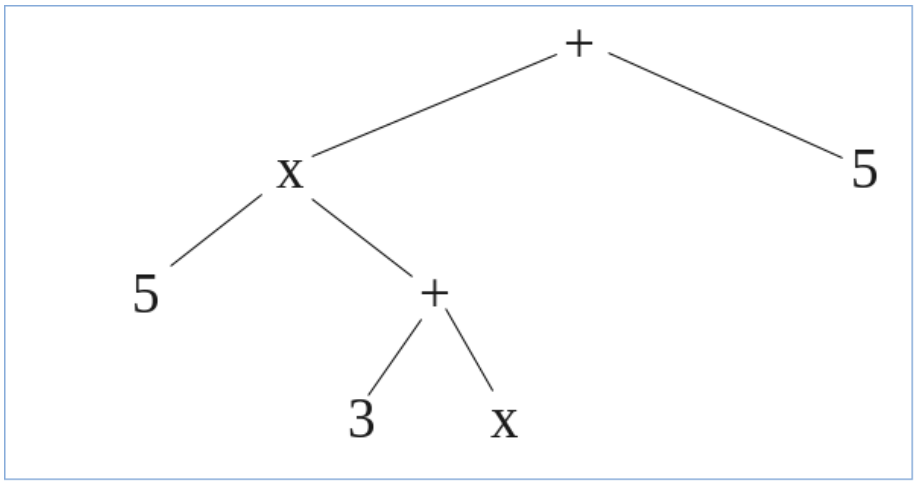
- $< E_1, \sigma > \twoheadrightarrow m_1$
- $< E_2, \sigma > \twoheadrightarrow m_2$

allora definiamo induttivamente, posto $m = m_1 + m_2$

$$< E_1 + E_2, \sigma > \twoheadrightarrow m$$

Esempio di calcolo
Mostriamo come calcolare

$$< 3 \times x + 5, \sigma > \twoheadrightarrow m \qquad \text{con} \qquad \sigma(x) = 7$$



quindi posto $E_1 = 3 \times x$ ed $E_2 = 5$ troviamo che

$$< 3 \times x, \sigma > \twoheadrightarrow m_1$$

$$< 5, \sigma > \twoheadrightarrow m_2$$

Quello che stiamo facendo in questi passaggi è costruire un albero di dimostrazione. Quindi troviamo subito che $m_2 = 5$, dobbiamo ora calcolare m_1, ma sappiamo

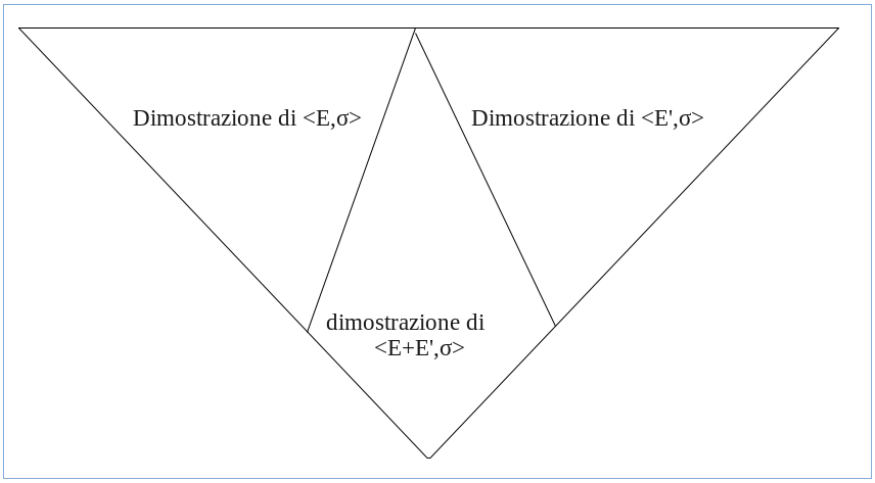
$$< 3, \sigma > \twoheadrightarrow 3$$

$$< x, \sigma > \twoheadrightarrow 7$$

quindi $m_1 = 7 \times 3 = 21$, quindi $m = 26$, pertanto

$$< 3 \times x + 5, \sigma > \twoheadrightarrow 26$$

Osserviamo che se ho le dimostrazioni di E ed E' posso comporle e ottenere la dimostrazione di $E + E'$



Dove per dimostrazione di $< E, \sigma >$ si intende $< E, \sigma > \twoheadrightarrow n$

Teorema di equivalenza delle semantiche

Vale che

$$\mathcal{A} = \mathcal{N}$$

prima parte della dimostrazione
Per dimostrare il teorema dobbiamo provare che

$$\mathcal{A}[\![E]\!]_\sigma = n \iff \mathcal{N}[\![E]\!]_\sigma$$

Mostriamo l'implicazione verso destra $\Rightarrow$

Per i casi base la dimostrazione è semplice, usando $< n, \sigma > \twoheadrightarrow n$ abbiamo

$$\mathcal{A}[\![n]\!]_\sigma = n \Rightarrow \mathcal{N}[\![n]\!]_\sigma = n$$

quindi un caso base è dimostrato, l'altro si dimostra allo stesso modo sostituendo il numero n con la variabile x e ricordando che $< x, \sigma > \twoheadrightarrow < \sigma(x) >$ e $\mathcal{A}[\![x]\!]_\sigma = \sigma(x)$, resta da mostrare il passo induttivo

Per definizione sappiamo che

$$\mathcal{A}[\![E_1 + E_2]\!]_\sigma = \mathcal{A}[\![E_1]\!]_\sigma \text{ più } \mathcal{A}[\![E_1 + E_2]\!]_\sigma$$

qundi se $< E_1, \sigma > \twoheadrightarrow n_1$ e $< E_2, \sigma > \twoheadrightarrow n_2$ allora posto $n = n_1 + n_2$ avremo sempre per definizione

$$\mathcal{A}[\![E_1 + E_2]\!]_\sigma = n$$

Daltronde

$$\mathcal{N}[\![E_1 + E_2]\!]_\sigma = n$$

vuol dire proprio

$$< E_1, E_2, \sigma > \twoheadrightarrow n$$

Quindi abbiamo provato che

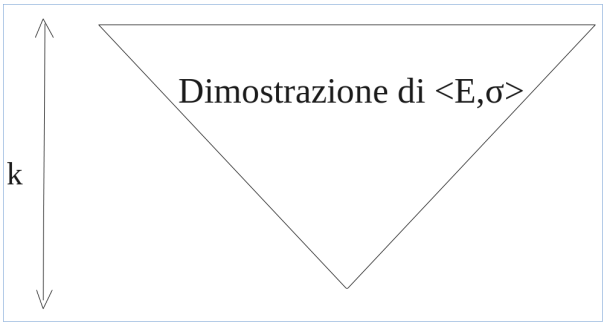
$$\mathcal{A}[\![E_1 + E_2]\!]_\sigma = n \Rightarrow \mathcal{N}[\![E_1 + E_2]\!]_\sigma = n$$

Per l'altra implicazione, ovvero $\Leftarrow$, dobbiamo mostrare che

$$< E, \sigma > \twoheadrightarrow n \Rightarrow \mathcal{A}[\![E]\!]_\sigma = n$$

Per fare questa dimostrazione possiamo fare induzione sulla lunghezza k della dimostrazione

$$< E, \sigma > \twoheadrightarrow n$$



dove osserviamo che se E è un assioma allora $k = 0$, per fare questa dimostrazione è necessario usare una sorta di induzione che è data dalla seguente proposizione

Proposizione sulla profondità della deduzione

Se $\mathcal{P}$ vale per le dimostrazioni di profondità zero, nell'ipotesi in cui il fatto che $\mathcal{P}$ valga per le dimostrazioni di profondità k implica che $\mathcal{P}$ vale per le dimostrazioni di profondità $k + 1$ allora $\mathcal{P}$ vale sempre.

seconda parte della dimostrazione
siamo ora in grado di concludere la dimostrazione del teorema. Per gli assiomi (profondità zero) abbiamo

$$< n, \sigma > \twoheadrightarrow m \Rightarrow \mathcal{A}[\![n]\!]_\sigma = m$$

e questa vale per definizione e come sempre lo stesso discorso si ripete sull'altro caso base, ovvero sulle variabili (la x).
Facciamo ora il passon induttivo, per ipotesi induttiva abbiamo

$$t =< E, \sigma > \twoheadrightarrow m \Rightarrow \mathcal{A}[\![E]\!]_\sigma = m$$

dove la dimostrazione di t è profonda al massimo k.
Chiaramente sono necessari i sottocasi sulla forma di E.

$$E = E_1 + E_2$$

il primo sottocaso è $< E, \sigma > \twoheadrightarrow n$ con una dimostrazione profonda $k+1$, questa è stata dedotta dalle premesse

- $< E_1, \sigma > \twoheadrightarrow m_1$
- $< E_2, \sigma > \twoheadrightarrow m_2$

ponendo $n = m_1$ più m_2, usando l'ipotesi induttiva

- $< E_1, \sigma > \Rightarrow \mathcal{A}[\![E_1]\!]_\sigma = m_1$
- $< E_2, \sigma > \Rightarrow \mathcal{A}[\![E_2]\!]_\sigma = m_2$

quindi per definizione

$$\mathcal{A}[\![E_1]\!]_\sigma \text{ più } \mathcal{A}[\![E_2]\!]_\sigma = \mathcal{A}[\![E_1 + E_2]\!]_\sigma = n$$

che è appunto la conclusione del teorena.

Semantica operazionale strutturale

Definiamo una nuova semantica strutturale

$$\mathcal{S} : \text{ Espressioni } \times \text{ Store } \rightarrow \mathcal{Z}$$

dove definiamo

$$\mathcal{S}[\![E]\!]_\sigma = n \quad \Longleftrightarrow \quad < E, \sigma > \xrightarrow{*} < n, \sigma >$$

chiaramente dobbiamo ancora definire l'operatore $\xrightarrow{*}$

Sistema di transizioni

Si consideri $< \Gamma, \rightarrow, F >$ dove

- Γ è l'insieme delle configurazioni
- $\rightarrow \subseteq \Gamma \times \Gamma$ è una relazione
- $F \subseteq \Gamma$ è l'insieme degli stati finali

Quindi $\xrightarrow{*}$ è la chiusura transitiva e riflessiva di $\rightarrow$, definiamo ora tutti questi oggetti. L'insieme delle configurazioni sarà

$$\Gamma = \{< E, \sigma > \quad \text{tale che} \quad \sigma : V \rightarrow \mathbb{Z} \quad \text{con} \quad V \supseteq Var(E)\}$$

mentre l'insieme degli stati finali sarà della forma

$$F = \{< n, \sigma >\}$$

Quindi il problema resta quello di definire esplicitamente $\rightarrow$. Definiamo come assioma

$$< x, \sigma > \rightarrow n \ \text{ se } \ \sigma(x) = m$$

Come definiamo la somma di due espressioni?

$$< E_1 + E_2, \sigma > \rightarrow \ ?$$

Osservazione
Con la semantica che abbiamo studiato prima, con l'approccio delle dimostrazioni, se avevamo una dimostrazione con due sottodimostrazioni non importava l'ordine con cui facevamo le sottodimostrazioni, potevamo fare prima l'una e poi l'altra purchè leggevamo tutto l'albero. Ora invece avremo un ordine.

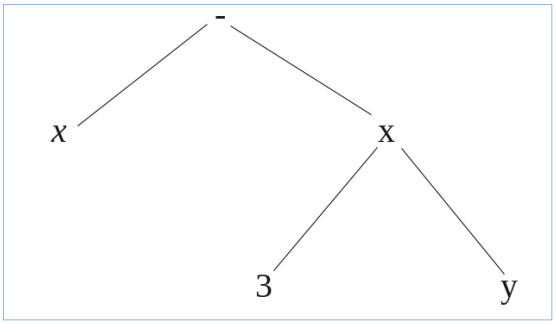
Quindi definiamo gli altri casi induttivi come

- $< E_1 + E_2, \sigma > \rightarrow < E_1' + E_2, \sigma > \quad$ dove $\quad < E_1, \sigma > \rightarrow < E_1', \sigma >$
- $< n + E_2, \sigma > \rightarrow < n + E_2', \sigma > \quad$ dove $\quad < E_2, \sigma > \rightarrow < E_2', \sigma >$
- $< n_1 + n_2, \sigma > \rightarrow < n_1 + E_2', \sigma > \quad$ dove $\quad < E_2, \sigma > \rightarrow < E_2', \sigma >$

A volte capiterà di sottointendere σ.

Esempio di calcolo

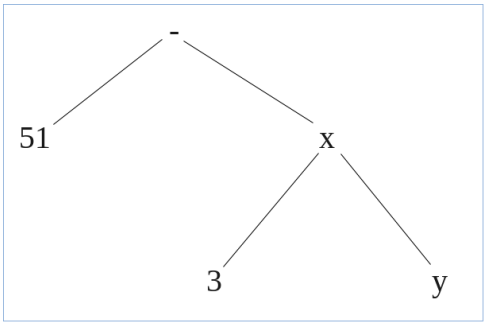
Consideriamo $x - 3 \times y$, allora usiamo la seguente notazione



$$\frac{\text{premessa}}{\text{passo}}$$

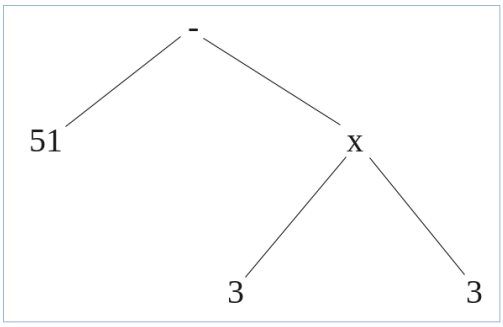
quindi avremo al primo passo

$$\frac{x \to \sigma(x) = 51}{x - 3 \times y \to 51 + (3 \times y)}$$



Facciamo ancora un altro passo sulla partentesi interna

$$\frac{y \to \sigma(y) = 3}{3 \times y \to 3 \times 3}$$



quindi abbiamo

$$\frac{3 \times y \to 3 \times 3}{51 - 3 \times y \to 51 - 3 \times 3}$$

Quindi la risposta è 42. Quindi in generale avremo una successione

$$\gamma_0 \xrightarrow{\nabla} \gamma_1 \xrightarrow{\nabla} \dots \xrightarrow{\nabla} \gamma_n$$

Dove ad ogni passo corrisponde un albero)

Teorema di equivalenza delle semantiche

Vale il risultato più generale

$$\mathcal{A} = \mathcal{S} = \mathcal{N}$$

Lemma di progresso

Per ogni coppia E e σ possono succedere due cose

- $E = n$
- esiste un E' tale che $< E, \sigma > \rightarrow < E', \sigma >$

ovvero le computazioni vanno avanti.

dimostrazione
Faccimo induzione sulla struttura di E.
Se $E = n$ non c'è nulla da dimostrare, se $E = x$ allora prendo come $E' = n = \sigma(x)$ e quindi la tesi.
Quindi i casi base sono fatti ora per ipotesi induttiva abbiamo che:

$$\text{se} \quad E = E_1 + E_2 \quad \text{allora} \quad \exists \quad E_1' \quad \text{tale che} \quad < E_1, \sigma > \rightarrow < E_1', \sigma >$$

allora $< E, \sigma > \rightarrow < E_1' + E_2, \sigma >$

Lemma di unicità del risultato

Se $E \rightarrow E'$ ed $E \rightarrow E''$ allora $E' = E''$

dimostrazione
Per i casi base non c'è nulla da dimostrare.
Se $E = E_1 + E_2$ allora

$$\frac{E_1 \rightarrow E_1'}{E \rightarrow E_1 = E_1' + E_2}$$

per ipotesi induttiva ho

$$E_1 \rightarrow E_1'' \Rightarrow E_1' = E_1''$$

quindi la tesi, per gli altri casi si fa in modo simile, ad esempio non è stato trattato

$$\frac{E_2 \rightarrow E_2'}{n + E_2 \rightarrow n + E_2'} \quad \text{con} \quad m_1 + m_2 \rightarrow n$$

Per completare la dimostrazione del teorema di equivalenza delle semantiche manca

$$\mathcal{N} = \mathcal{S}$$

ovvero devo mostrare

$$E \twoheadrightarrow n \iff E \xrightarrow{*} n$$

dimostrazione
Una implicazione è facile, ovvero $\Rightarrow$, basta fare induzione sulla profondità della dimostrazione. Per l'altra implicazione $\Leftarrow$ dobbiamo fare induzione sulla lunghezza della computazione.

Quindi piuttosto che usare $[]*$ useremo i passi contati $\xrightarrow{k}$.
Inoltre sarà necessario usare l'induzione nel seguente modo.

Se $\mathcal{P}$ vale per $\gamma \xrightarrow{0} \gamma$ e se il fatto che $\mathcal{P}$ valga per $\gamma \xrightarrow{k} \gamma$ implica che $\mathcal{P}$ vale per $\gamma \xrightarrow{k+1} \gamma$, allora $\mathcal{P}$ vale sempre per ogni γ.

Lemma

Se $< E, \sigma > \xrightarrow{k} m_1$ allora per ogni E_2 vale che $< E_1 + E_2, \sigma > \xrightarrow{k} < m_1 + E_2, \sigma >$