Linguaggio CICLO (o LOOP) – L

(introdotto nel 1967 da A. Meyer e D.M. Ritchie)

La sintassi di L è del tutto simile a quella del linguaggio S .

Le istruzioni base del linguaggio sono:

1. **V ← 0**
2. **V ← V+1**
3. **V ← V′**
4. **LOOP V**
5. **END**

Le istruzioni (4) e (5) vanno sempre in coppia e devono essere associate come le parentesi aperte e chiuse.
Il linguaggio L ha alcune caratteristiche significative:

1. Calcola solo funzioni ovunque definite.
2. I programmi di L calcolano tutte e sole le funzioni ricorsive primitive;
3. E' possibile introdurre, in modo naturale, una nozione di ***complessità*** di calcolo.

ESEMPI

```
Z ← X1
LOOP X2
   Z ←Z+1
END
Y ←Z
```

(SOMMA)

```
1. Z ← 0
2. LOOP X1
3.       LOOP X2
4.             Z ← Z+1
5.       END
6. END
7. Y ← Z
```

(PRODOTTO)

Le istruzioni LOOP-END possono essere inserite all'interno di altre coppie LOOP-END nel modo visto prima.

Tale processo si chiama *nidificazione*.
Si può introdurre il concetto di "profondità di nidificazione" delle istruzioni LOOP-END per misurare il numero di volte in cui una coppia LOOP-END compare all'interno di altre coppie LOOP-END.

Diciamo che un programma ha profondità di nidificazione 1 se la coppia di istruzioni LOOP-END compare in forma semplice, ossia se il blocco contenuto tra l'istruzione LOOP e l'istruzione END non contiene istruzioni LOOP-END.

In generale diciamo che un programma ha profondità di nidificazione n se vi è almeno una coppia LOOP-END tale che il blocco di istruzioni che contiene al suo interno contiene almeno una coppia nidificata n – 1 volte.

I programmi con profondità di nidificazione 0 sono quelli che non contengono istruzioni LOOP-END.

Che succede al crescere della *profondità di nidificazione*?

Succede che si possono calcolare nuove funzioni e queste funzioni sono più complesse di quelle calcolate prima nello stesso senso in cui il prodotto è più complesso della somma.

Estensione

Il linguaggio WHILE (W)

Aggiungendo alle istruzioni di L la seguente istruzione:

WHILE X ≠ 0 DO

END

Otteniamo un linguaggio più potente che calcola tutte le funzioni calcolabili in S

Sintesi dei modelli di computo introdotti

AL MASSIMO LIVELLO DI GENERALITA'

Macchina di Turing

Linguaggio S

Funzioni μ–ricorsive

Linguaggi Sn

Linguaggi di Post –Turing

Linguaggio WHILE

CALCOLANO UNA SOTTOCLASSE:

Funzioni ricorsive primitive

Linguaggio CICLO

OPERAZIONE DI DECREMENTO NEL LINGUAGGIO LOOP

```
Z ← 0
LOOP X
  X ← Z
  Z ← Z+1
END
Y ← X
```

$$X \leftarrow X-1$$

Il modo in cui tale coppia di istruzioni funziona forse è chiarito meglio mediante esempi.
Qui ci limitiamo a dire che esse fanno ripetere, ciclare (loop = ciclo) ciò che è contenuto tra l'istruzione LOOP e l'istruzione END.
Ciò fatto, verrà immediatamente eseguita l'istruzione che segue END.

Osservazione:
Il numero di volte che il blocco di istruzioni contenuto tra LOOP e END deve essere eseguito è dato dal valore della variabile X dopo LOOP nel momento in cui l'istruzione LOOP è incontrata.
Quindi anche se il valore di X viene modificato nel corso del calcolo, ciò non ha alcuna influenza nel numero di volte che deve essere ripetuto il "ciclo".

Conseguenza di ciò
Le istruzioni LOOP-END non possono perciò indurre meccanismi di non terminazione.
Poiché anche le altre istruzioni del linguaggio LOOP $V \leftarrow 0$, $V \leftarrow V+1$, $V \leftarrow V'$ non possono farlo, possiamo concludere che il linguaggio di programmazione L, a differenza del linguaggio $\mathcal{L}$, non dà la possibilità di scrivere programmi che non terminano.
Tutti i programmi di L prima o poi si fermano.
Quindi se un programma, sintatticamente corretto, di L non si è fermato possiamo essere sicuri che non ha finito di calcolare e che dobbiamo ancora attendere (non inutilmente).

Esempio

Consideriamo il programma

1. $X \leftarrow 0$
2. $X \leftarrow X+1$
3. LOOP X
4. $X \leftarrow X+1$
5. END
6. $Y \leftarrow X$

Cosa calcola questo programma?

La funzione costante $f(x) = 2$.

Infatti quando viene attivata l'istruzione 3 la X ha il valore 1.
La 3 deve quindi leggersi "ciclo una volta"
(e questo è indipendente dal fatto che la 4, a sua volta, aumenta il valore della stessa variabile X)
La 4 viene perciò eseguita una sola volta ed il valore finale di X è, perciò, 2.
(indipendentemente dal valore d'ingresso di X, perché l'istruzione 1 provvede ad azzerarlo)
Il valore 2 di X viene, infine, mediante l'istruzione 6, assegnato alla variabile di uscita Y.

Consideriamo ora un altro programma di L

```
1. Z ← 0
2. LOOP X1
3.     LOOP X2
4.         Z ← Z+1
5.     END
6. END
7. Y ← Z
```

Cosa fa questo programma?
L'istruzione 2 ci dice di ripetere X_1 volte quello che è compreso tra la 2 e la 6, cioè il blocco di istruzioni 3-4-5.
A sua volta l'istruzione 3 dice di ripetere X_2 volte ciò che fa l'istruzione 4.
Ma l'istruzione 4 non fa altro che aggiungere una unità alla variabile ausiliaria Z (che è inizialmente azzerata)
Quindi l'effetto del blocco di istruzioni 3-4-5 è quello di incrementare il valore di Z di X_2 unità.
A sua volta l'istruzione 2 prescrive di ripetere questo procedimento X_1 volte.
Ma aumentare il valore di Z di X_2 unità per X_1 volte significa aumentarlo di $X_1 \cdot X_2$ unità.
Poiché Z era inizialmente 0 e l'istruzione 7 assegna il valore di Z alla variabile di uscita, il programma non fa altro che calcolare

L'esempio precedente ha mostrato che le istruzioni LOOP-END possono essere inserite all'interno di altre coppie LOOP-END.
Tale processo si chiama nidificazione.

Si può quindi introdurre il concetto di "profondità di nidificazione" delle istruzioni LOOP-END per misurare il numero di volte in cui una coppia LOOP-END compare all'interno di altre coppie LOOP-END.

Diciamo che un programma ha profondità di nidificazione 1 se la coppia di istruzioni LOOP-END compare in forma semplice, cioè se il blocco contenuto tra l'istruzione LOOP e l'istruzione END non contiene istruzioni LOOP-END.

In generale diciamo che un programma ha profondità di nidificazione n se vi è almeno una coppia LOOP-END tale che il blocco di istruzioni che contiene al suo interno contiene almeno una coppia nidificata n-1 volte.

I programmi con profondità di nidificazione 0 sono quelli che non contengono istruzioni LOOP-END.

I programmi che abbiamo considerato hanno profondità di nidificazione 1 e 2 rispettivamente.

In che modo può essere utilizzata tale nozione di "profondità di nidificazione"?

Riprendiamo in esame l'esempio 2 con il programma per la moltiplicazione (che ha profondità di nidificazione 2)
Quale potrebbe essere una ne semplificazione che permetta di calcolare la somma?
Ad es. il programma seguente:

```
Z ← X1
  LOOP X2
    Z ← Z + 1
  END
Y ← Z
```

La profondità di nidificazione di tale programma è 1.
Si ha la sensazione che, aumentando la profondità di nidificazione, si possano fare, naturalmente, cose via via più complesse.

Renderemo successivamente precise tali considerazioni, per il momento introduciamo semplicemente i seguenti linguaggi:

Sia L_n la classe dei programmi-ciclo con coppie LOOP-END nidificate fino ad una profondità al più n e sia, in corrispondenza, $\mathcal{L}_n$ la classe delle funzioni calcolabili dai programmi di L_n.

In base alla terminologia introdotta, L_0 è la classe di programmi che non contengono istruzioni LOOP-END, il programma per l'addizione appartiene ad L_1, quello per la moltiplicazione ad L_2.

La classe di tutte le funzioni calcolabili mediante programmi-ciclo è quindi data da $\bigcup_{n=0}^{\infty} L_n$.

Proposizione

Le funzioni ricorsive primitive appartengono alla classe $\mathcal{L} = \bigcup_{n=0}^{\infty} \mathcal{L}_n$ delle funzioni calcolabili dai programmi-ciclo.

Dimostrazione

È sufficiente mostrare che le funzioni iniziali appartengono ad $\mathcal{L}$ e che $\mathcal{L}$ è chiusa rotto le operazioni di composizione e ricorsione.

Si mostra immediatamente che le funzioni iniziali appartengono ad $\mathcal{L}_0$.

- La funzione successore $s(n)$ è calcolata dal programma:

$$Z \leftarrow X$$
$$Z \leftarrow Z+1$$
$$Y \leftarrow Z$$

- Poiché tra le istruzioni di L vi è quella di assegnazione la funzione costante zero ($u(x)$) e le funzioni di selezione sono calcolate dai seguenti programmi di una sola istruzione:

$u(x)$	$u_i^n(x_1, \ldots, x_n)$
$Y \leftarrow 0$	$Y \leftarrow X_i$

Mostriamo adesso che l'operazione di composizione applicata a funzioni calcolabili da programmi-ciclo dà luogo a funzioni sempre calcolabili da programmi ciclo.

Un programma-ciclo che calcola la funzione $f(g_1(x_1,\dots,x_m),\dots,g_n(x_1,\dots,x_m))$ dove la f e le g_i sono calcolabili da programmi-ciclo è dato infatti da:

$$Z_1 \leftarrow g_1(x_1,\dots,x_m)$$
$$\dots\dots\dots\dots\dots$$
$$Z_n \leftarrow g_n(x_1,\dots,x_m)$$
$$Y \leftarrow f(z_1,\dots,z_n)$$

Ogni riga del programma precedente è infatti una macro ammissibile nel nostro linguaggio perché in L è già disponibile una istruzione di assegnazione e le f, g_i sono ciclo-calcolabili.

Osservazione

Se k è la profondità massima di nidificazione delle macroespansioni del programma precedente allora k sarà anche la profondità massima di nidificazione dell'intero programma perché esso non introduce ulteriori coppie LOOP-END

LA COMPOSIZIONE DI FUNZIONI in $\mathcal{L}_k$ PRODUCE UNA FUNZIONE ANCORA IN $\mathcal{L}_k$.

Passiamo adesso all'operazione di ricorsione.

Consideriamo direttamente il caso più generale

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

con f e g cicle-calcolabili

Allora la funzione h è calcolata da:

```
Y ← f(X1,...,Xn)
Z ← 0
LOOP Xn+1
   Y ← g(Z, Y, X1,...,Xn)
   Z ← Z+1
END
```

Osserviamo che:
se k è la profondità di nidificazione di f ed in quella di g, allora la profondità di nidificazione di h sarà pari al massimo tra k ed $m+1$.

Abbiamo così dimostrato la proposizione.

Vogliamo mostrare adesso che le funzioni calcolabili da programmi-Ciclo sono ricorsive primitive. Ciò dimostra, assieme al risultato precedente, l'equivalenza tra le due nozioni.

Il nucleo di tale dimostrazione risiede nel mostrare che i meccanismi (algoritmici) di trasformazione dei valori delle variabili che si possono mettere in atto mediante le istruzioni del linguaggio LOOP sono tutti "simulabili" mediante meccanismi esprimibili all'interno della ricorsività primitiva.

Esamineremo ora le variazioni indotte sui valori delle variabili da un programma-Ciclo in modo del tutto generale.

In particolare faremo le due assunzioni seguenti:

- considereremo solo variabili locali
- assumeremo che le istruzioni contenute tra un LOOP ed un END non contengano mai la variabile che compare dopo LOOP.

  Questo non comporta alcuna restrizione perché, ovviamente, si può sempre operare il seguente cambio di variabili:

Immaginiamo allora di avere un programma P di L, le variabili di L avranno dei valori assegnati prima di far girare P e avranno degli altri valori dopo che P si sarà fermato.

Possiamo quindi pensare a P come ad una macchinetta che effettua tale trasformazione nelle variabili.

Se le variabili che compaiono in P sono $z_1, z_2, \ldots, z_n$ (tutte locali per l'assunzione fatta) allora tale trasformazione può essere rappresentata nel modo seguente:

$$z_1 \leftarrow f_1(z_1, \ldots, z_n)$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$z_n \leftarrow f_n(z_1, \ldots, z_n)$$

Immaginiamo adesso di considerare il programma P come il blocco interno da ciclare di un'istruzione LOOP-END:

LOOP V
[P]
END

dove V, in base alla seconda assunzione fatta non compare in P. Sia Q tale nuovo programma.

Q indurrà, a sua volta, la seguente trasformazione sulle $n+1$ variabili $z_1, z_2, \ldots, z_n, V$:

$$z_1 \leftarrow g_1(z_1, \ldots, z_n, V)$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$z_n \leftarrow g_n(z_1, \ldots, z_n, V)$$

Ci poniamo adesso le due domande seguenti:

1) Che rapporto esiste tra le f_i e le g_i?
2) È possibile trovare una caratterizzazione di tali funzioni?

La risposta è fornita dalle due proposizioni seguenti:

Proposizione 1

Se le funzioni $f_1, \dots, f_n$ sono ricorsive primitive allora lo sono anche le funzioni $g_1, \dots, g_n$.

Dimostrazione

Come facciamo a calcolare i valori della funzione g_i su $(z_1, \dots, z_n, t+1)$?

Andando a calcolare la corrispondente funzione f_i sui valori $g_1(z_1, \dots, z_n, t), \dots, g_n(z_1, \dots, z_n, t)$; cioè mediante il seguente meccanismo di ricorsione simultanea:

$$(*) \quad \begin{cases} g_i(z_1, \dots, z_n, 0) = z_i \\ g_i(z_1, \dots, z_n, t+1) = f_i(g_1(z_1, \dots, z_n, t), \dots, g_n(z_1, \dots, z_n, t)) \end{cases}$$

La scrittura precedente non ci consente di concludere immediatamente che le g_i sono ricorsive primitive perché la ricorsione non si presenta nella forma semplice che sappiamo che preserva la ricorsività primitiva.

Utilizziamo allora dei meccanismi di codifica.

Poniamo: $(**) \quad \tilde{g}(z_1, \dots, z_n, u) = [g_1(z_1, \dots z_n, u), \dots, g_n(z_1, \dots, z_n, u)]$

per cui si ha che:

$$\tilde{g}(z_1, \dots, z_n, 0) = [z_1, \dots, z_n]$$

e $$\tilde{g}(z_1, \dots, z_n, t+1) = [k_1, \dots, k_n]$$

dove i k_i sono ottenuti mediante la $(*)$ e $(**)$ cioè si ha:

$$k_i = f_i((\tilde{g}(z_1, \dots z_n, t))_1, \dots (\tilde{g}(z_1, \dots, z_n, t))_n)$$

In cosa differisce l'ultima scrittura che riguarda le $\tilde{g}$ dalla $(*)$?

In quest'ultimo caso (a differenza di (*)) abbiamo operazioni ricorsive primitive applicate a funzioni (f_i) che, per ipotesi, sono ricorsive primitive e quindi possiamo concludere che $\tilde{g}(z_1, \ldots, z_n, u)$ è ricorsiva primitiva.

Poiché si ha che

$$g_i(z_1, \ldots, z_n, u) = (\tilde{g}(z_1, \ldots, z_n, u))_i$$

la proposizione è dimostrata.

PROPOSIZIONE 2

Sia P un programma LOOP che contiene solo le variabili $z_1, \ldots, z_n$.

P trasforma i valori delle variabili $z_1, \ldots, z_n$ secondo lo schema:

$$z_1 \leftarrow f_1(z_1, \ldots, z_n)$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$z_n \leftarrow f_n(z_1, \ldots, z_n)$$

Allora le funzioni $f_1, \ldots, f_n$ sono tutte ricorsive primitive.

Dimostrazione

Per induzione. Assumiamo in primo luogo che sia $P \in L_0$. Allora P non può contenere istruzioni ciclo e le uniche istruzioni che può utilizzare sono:

- porre una variabile uguale a zero o uguale al valore di un'altra variabile
- aggiungere 1 a una variabile un numero finito di volte.

Quindi le f possono assumere soltanto una delle due forme:

- $f_i(x_1, \ldots, x_n) = x_j + k$
- $f_i(x_1, \ldots, x_n) = k$, per qualche k,

Queste funzioni sono ricorsive primitive.

Adesso ammettiamo che il risultato sia vero per i programmi di L_n, vogliamo mostrare che lo è anche per un arbitrario programma P di L_{n+1}.

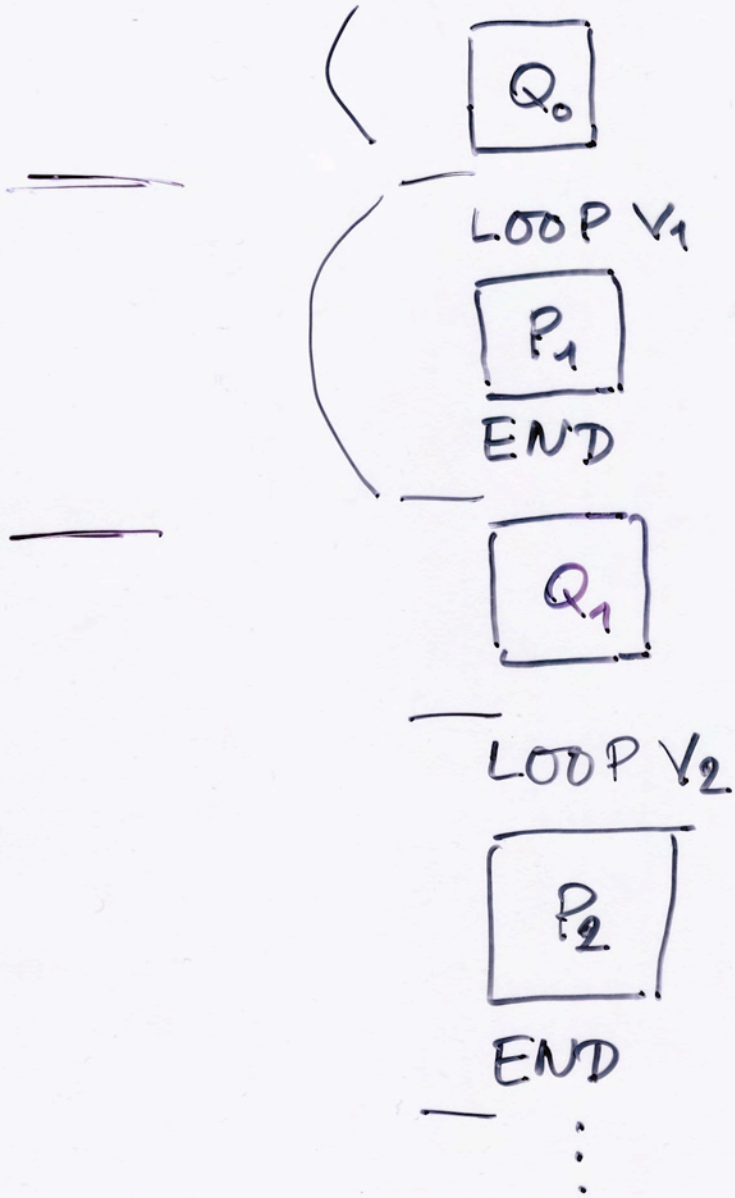
Un programma di L_{n+1} si può decomporre in una serie di blocchi successivi ciascuno dei quali forma un programma appartenente ad L_n, eventualmente inseriti in un ciclo LOOP-END suddividiamo con la lettera P_i questi ultimi e con le lettere Q_i gli altri.

Per l'ipotesi di induzione le funzioni calcolate da ciascun blocco sono quindi ricorsive primitive.

Ma in base alla proposizione 1 appena dimostrata, se la funzione calcolata da P_i è ricorsiva primitiva allora lo è anche quella calcolata da:

LOOP V_i
P_i
END

Possiamo allora concludere che la proposizione è dimostrata perché rimane solo da applicare un'operazione di composizione che preserva notoriamente la ricorsività primitiva.


Q_0
LOOP V_1
P_1
END
Q_1
LOOP V_2
P_2
END

Osservazione

Un altro modo, forse più diretto, di dimostrare il risultato è quello di considerare una codifica del programma complessivo, ma questo avrebbe comportato riadattare tutto l'apparato di codifica, sviluppato per $\mathcal{S}$, al linguaggio L.

Abbiamo adesso tutti gli elementi per dimostrare che i programmi-ciclo calcolano funzioni ricorsive primitive.

Sia P un programma di L che calcola la funzione $h(x_1, \ldots, x_k)$.
P può essere trasformato nel programma

$$Z_1 \leftarrow X_1$$
$$\cdots\cdots\cdots$$
$$Z_k \leftarrow X_k$$
$$Q$$
$$Y \leftarrow Z_s$$

che è una trasformazione del tipo di quelle che abbiamo discusso, Q contiene solo variabili locali $Z_1, \ldots, Z_m$ con $k < s \leq m$.
Si ha che:

$$h(x_1, \ldots, x_k) = f_s(x_1, \ldots, x_k, 0, \ldots, 0)$$ e

poiché f_s è ricorsiva primitiva lo è anche h.

Passiamo adesso all'altro aspetto del linguaggio ciclo e cioè quello di fornire uno strumento per definire una prima misura della complessità di calcolo.

Come prima misura di complessità prenderemo il tempo di calcolo di tali programmi e, a sua volta, tale tempo di calcolo verrà definito nella maniera più semplice possibile e cioè come il numero totale di istruzioni di assegnazione (zero o altro valore) e di incremento che sono eseguite.

Dunque, se P è un programma-ciclo con variabili di ingresso $x_1, \ldots, x_n$ allora $T_P(x_1, \ldots, x_n)$ è il tempo di calcolo di P, definito nel modo precedentemente detto.

FATTO:
Esiste un programma che calcola T_P e che ha una profondità di iterazione non maggiore di P.
In simboli: Se $P \in L_n$ allora $T_P \in L_n$

Dimostrazione.
Basta modificare il programma P inserendo un contatore T che aumenta di uno ogni volta che viene eseguita una delle istruzioni $V \leftarrow 0$, $V \leftarrow V'$, $V \leftarrow V+1$.
Questo può realizzarsi ponendo un'istruzione $T \leftarrow T+1$ subito dopo ciascuna istruzione del tipo precedente presente in P.
È chiaro che questo nuovo programma ha la stessa profondità di iterazione di P.

Vogliamo adesso limitare superiormente i tempi di calcolo di varie funzioni.

Ricordiamo la notazione:

$$g^{(n)} = \underbrace{g(g(\dots g(n)))}_{n \text{ volte}}, \qquad g^{(0)}(n) = n$$

(composizione di g con se stessa n volte)

Definiamo adesso la seguente famiglia di funzioni:

$$f_0(n) = \begin{cases} n+1 & \text{se } n=0 \text{ oppure } n=1 \\ n+2 & \text{altrimenti} \end{cases}$$

$$f_{n+1}(n) = f_n^{(n)}(1)$$

Si ha che:

$$f_1(n) = 2n \qquad (\text{con } n \neq 0)$$

$$f_2(n) = 2^n$$

$$f_3(n) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \quad (n \text{ volte})$$

Vogliamo studiare alcune proprietà di questa famiglia di funzioni.

$$f_1(x) = f_0^{(x)}(1) = \underbrace{f_0(f_0 \dots f_0}_{x \text{ volte}}(1)\dots)$$

ma $f_0(1) = 2$

ed ogni volta successiva l'applicazione di f_0 applicherà 2; questo lo fa per x volte, quindi $f_1(x) = 2x$

(per $x \neq 0$)

Analogamente

$$f_2(x) = f_1^{(x)}(1) = \underbrace{f_1(f_1 \dots f_1}_{x \text{ volte}}(1)\dots)$$

$f_1(1) = 2$, ad ogni passo successivo l'applicazione di f_1 causa una moltiplicazione per 2, quindi $f_2(x) = 2^x$.