LINGUAGGIO DI PROGRAMMAZIONE L

Il **linguaggio Loop** (o ciclo) presenta due caratteristiche significative:

1. I programmi di L calcolano esattamente le funzioni ricorsive primitive.
2. È possibile introdurre una nozione di complessità di calcolo.

Le istruzioni base di L sono:

- **V ← 0**
- **V ← V+1**
- **V ← V'**
- **LOOP V**
- **END**

Le ultime due istruzioni vanno sempre in coppia e devono essere associate come le parentesi aperte e chiuse. Queste fanno ripetere ciò che è contenuto tra l'istruzione LOOP e l'istruzione END. Dopodiché verrà immediatamente eseguita l'istruzione che segue END. 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 sul numero di volte che deve essere ripetuto il ciclo. Di conseguenza, le istruzioni LOOP – END, come anche le altre istruzioni del linguaggio L, non possono indurre meccanismi di non terminazione; quindi il linguaggio di programmazione L non dà la possibilità di scrivere programmi che non terminano.

Esempio 1): consideriamo il programma

1. X ← 0
2. X ← X+1
3. LOOP X
4. X ← X+1
5. END
6. Y ← X

Questo programma calcola la funzione costante f(x) = 2.

Esempio 2): consideriamo il programma

1. Z ← 0
2. LOOP X_1
3. LOOP X_2
4. Z ← Z+1
5. END
6. END
7. Y ← Z

Questo programma calcola il prodotto fra X_1 e X_2 e lo assegna alla variabile di uscita Y.

La nidificazione consiste nell'inserire istruzioni LOOP – END all'interno di altre coppie LOOP – END. Introduciamo 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. Un programma ha profondità di nidificazione 1 se il blocco contenuto tra l'istruzione LOOP e l'istruzione END non contiene istruzioni LOOP – END. In generale, 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, ha 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 nei due esempi precedenti hanno rispettivamente profondità di nidificazione 1 e 2.

Programma che calcola la somma fra X_1 e X_2:

1. $Z \leftarrow X_1$
2. LOOP X_2
3. $Z \leftarrow Z+1$
4. END
5. $Y \leftarrow Z$

La profondità di nidificazione di tale programma è 1.

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.
L_0 è la classe di programmi che non contengono istruzioni LOOP – END.
La classe di tutte le funzioni calcolabili mediante programmi-ciclo è quindi data da $\bigcup_{n=0}^{\infty} \mathcal{L}_n$

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

DIM: Bisogna dimostrare che le funzioni iniziali appartengono ad $\mathcal{L}$ e che $\mathcal{L}$ è chiusa sotto le operazioni di composizione e ricorsione.
Le funzioni iniziali appartengono a $\mathcal{L}_0$ e sono calcolate dai seguenti programmi:

Successore S(x)	**Costante zero n(x)**	**Selezione $u_i^n(x_1, \dots, x_n)$**
$Z \leftarrow X$ $Z \leftarrow Z+1$ $Y \leftarrow Z$	$Y \leftarrow 0$	$Y \leftarrow X_i$

Mostriamo 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 da

$$
\begin{gathered}
Z_1 \leftarrow g_1(X_1, \dots, X_m) \\
\dots \dots \dots \dots \dots \dots \dots \dots \\
Z_n \leftarrow g_n(X_1, \dots, X_m) \\
Y \leftarrow f(Z_1, \dots, Z_n)
\end{gathered}
$$

Il programma non introduce ulteriori coppie LOOP – END e quindi la composizione di funzioni in $\mathcal{L}_k$ produce una funzione ancora in $\mathcal{L}_k$.

Mostriamo che l'operazione di **ricorsione** applicata a funzioni calcolabili da programmi-ciclo dà luogo a funzioni sempre calcolabili da programmi-ciclo:
un programma-ciclo che calcola la funzione
{ $h(x_1, \dots, x_n, 0) = f(x_1, \dots, x_n)$
{ $h(x_1, \dots, x_n, z+1) = g(z, h(x_1, \dots, x_n, z), x_1, \dots, x_n)$
con f e g ciclo-calcolabili, è dato da

$Y \leftarrow f(X_1, \dots, X_n)$
$Z \leftarrow 0$
LOOP X_{n+1}
$Y \leftarrow g(Z, Y, X_1, \dots, X_n)$
$Z \leftarrow Z+1$
END

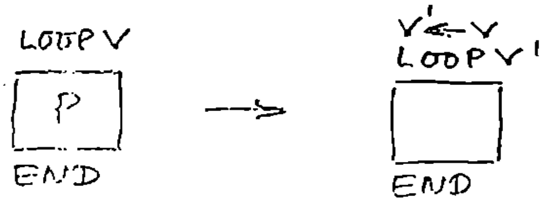
Il programma introduce un'istruzione LOOP – END quindi se k è la profondità di nidificazione di f, ed m quella di g, allora la profondità di nidificazione di h sarà pari al massimo tra k ed (m+1).

Le funzioni calcolabili da programmi-ciclo sono ricorsive primitive.

Assumiamo che:

1) le variabili considerate sono solo variabili locali;
2) le istruzioni contenute tra un LOOP e un END non contengono mai la variabile che compare dopo LOOP.

Questo non comporta alcuna restrizione poiché possiamo operare nel modo seguente:



Immaginiamo 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. Se le variabili che compaiono in P sono $z_1, \dots, z_n$ (tutte locali) allora possiamo pensare che P effettui la seguente trasformazione:

$$Z_1 \leftarrow f_1(Z_1, \dots, Z_n)$$
$$\dots \dots \dots \dots \dots \dots \dots$$
$$Z_n \leftarrow f_n(Z_1, \dots, Z_n)$$

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

LOOP V
P (dove V non compare in P)
END

Sia Q tale nuovo programma, esso indurrà a sua volta sulle n+1 variabili $Z_1, \dots, Z_n, V$ la seguente trasformazione:

$$Z_1 \leftarrow g_1(Z_1, \dots, Z_n, V)$$
$$\dots \dots \dots \dots \dots \dots \dots$$
$$Z_n \leftarrow g_n(Z_1, \dots, Z_n, V)$$

PROPOSIZIONE 1): Se le funzioni $f_1, \dots, f_n$ sono ricorsive primitive allora anche $g_1, \dots, g_n$ sono funzioni ricorsive primitive.

DIM: Per calcolare i valori della funzione g_i su $(z_1, \dots, z_n, t+1)$, calcoliamo la corrispondente funzione f_i sui valori $g_1(z_1, \dots, z_n, t), \dots, g_n(z_1, \dots, z_n, t)$ mediante il seguente meccanismo di ricorsione simultanea:

$\{ g_i(z_1, \dots, z_n, 0) = z_i$

$\{ g_i(z_1, \dots, z_n, t+1) = f_i(g_1(z_1, \dots, z_n, t), \dots, g_n(z_1, \dots, z_n, t))$

Questa scrittura però non ci consente di stabilire che le g sono ricorsive primitive perché la ricorsione non si presenta nella forma semplice. Quindi poniamo

$\breve{g}(z_1, \dots, z_n, u) = [g_1(z_1, \dots, z_n, t), \dots, g_n(z_1, \dots, z_n, t)]$ per cui si ha che

$\breve{g}(z_1, \dots, z_n, 0) = [z_1, \dots, z_n]$ e $\breve{g}(z_1, \dots, z_n, t+1) = [k_1, \dots, k_n]$ dove $k_i = f_i(\breve{g}(z_1, \dots, z_n, t)_1, \dots, \breve{g}(z_1, \dots, z_n, t)_n)$

A differenza della forma precedente, ora abbiamo operazioni ricorsive primitive applicate a funzioni (le f_i) che per ipotesi sono ricorsive primitive e quindi possiamo concludere che $\breve{g}(z_1, \dots, z_n, u)$ è ricorsiva primitiva. Poiché si ha che $g_i(z_1, \dots, z_n, u) = \breve{g}(z_1, \dots, z_n, u)_i$ la proposizione è dimostrata.

PROPOSIZONE 2): Sia P un programma LOOP che contiene solo le variabili $Z_1, \dots, Z_n$ e le trasformi secondo lo schema

$$Z_1 \leftarrow f_1(Z_1, \dots, Z_n, V)$$
$$\dots \dots \dots \dots \dots \dots \dots \dots$$
$$Z_n \leftarrow f_n(Z_1, \dots, Z_n, V)$$

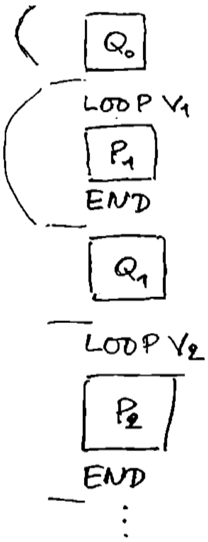
Allora le funzioni $f_1, \dots, f_n$ sono tutte ricorsive primitive.
DIM (per induzione): Ammettiamo che $P \in L_0$, quindi P non contiene istruzioni ciclo e le uniche istruzioni che può contenere sono:

- 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 solo una delle due forme:

- $f_i(z_1, \dots, z_n) = z_i+k$
- $f_i(z_1, \dots, z_n) = k$ per qualche k

Queste funzioni sono ricorsive primitive.
Ammettiamo che il risultato sia vero per i programmi L_n e dimostriamo 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. Indichiamo con P_i questi ultimi e con Q_i gli altri:



Per l'ipotesi di induzione le funzioni calcolate da ciascun blocco sono quindi ricorsive primitive.
Dalla proposizione 1) si ha che se la funzione calcolata da P_i è ricorsiva primitiva allora lo è anche quella calcolata da
LOOP V_i
P_i
END
Possiamo concludere che la proposizione è dimostrata perché rimane solo da applicare un'operazione di composizione che preserva la ricorsività primitiva.

Dimostriamo ora che <u>i programmi-ciclo calcolano funzioni ricorsive primitive</u>.
Sia P un programma di L che calcola la funzione $h(x_1, \dots, x_k)$. P può essere trasformato nel programma

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

dove Q contiene solo variabili locali $Z_1, \dots, Z_m$ con $k < s \leq m$.
Si ha che $h(x_1, \dots, x_k) = f_s(x_1, \dots, x_k, 0, \dots, 0)$ e poiché f_s è ricorsiva primitiva, lo è anche h.

Per definire una prima misura della complessità di calcolo prendiamo il tempo di calcolo di tali programmi, definito 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 $X_1, \dots, X_n$, allora $T_P(x_1, \dots, x_n)$ è il tempo di calcolo di P. Il programma che calcola T_P ha una profondità di nidificazione non maggiore di P:

$$\textbf{se } P \in L_n \textbf{ allora } T_P \in \mathcal{L}_n$$

DIM: Basta modificare il programma P inserendo un contatore T che aumenta di 1 ogni volta che viene eseguita un'istruzione di assegnazione e di incremento. Questo può realizzarsi ponendo un'istruzione $T \leftarrow T+1$ subito dopo ciascun istruzione del tipo $V \leftarrow 0$, $V \leftarrow V'$ oppure $V \leftarrow V+1$.
Questo nuovo programma ha la stessa profondità di nidificazione di P.

Introduciamo la seguente notazione:
$g^{(m)}(x) = g(g(\dots g(x)))$ composizione di g con se stessa m volte
$g^{(0)}(x) = x$

Definiamo adesso la seguente famiglia di funzioni:

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

Si ha che:
$f_1(x) = 2x$ (con $x \neq 0$)
$f_2(x) = 2^x$
$f_3(x) = 2^{2^\wedge 2^\wedge \dots 2 \text{ (x volte)}}$

<u>LEMMA 1</u>: $\mathbf{f_{n+1}(x+1) = f_n(f_{n+1}(x))}$
DIM: $f_{n+1}(x+1) = f_n^{(x+1)}(1) = f_n(f_n^{(x)}(1)) = f_n(f_{n+1}(x))$

<u>LEMMA 2</u>: $\mathbf{f_0^{(k)}(x) \geq k}$
DIM (per induzione su k): Per $k = 0$ si ha $f_0^{(0)}(x) = x \geq 0$. Assumiamo il risultato valido per k e dimostriamo che è valido per k+1:
- $f_0^{(k+1)}(x) = f_0(f_0^{(k)}(x))$
- poiché $f_0(x) \geq x+1$ per ogni x (per definizione di f_0) si ha che $f_0(f_0^{(k)}(x)) \geq f_0^{(k)}(x)+1$
- poiché per l'ipotesi di induzione $f_0^{(k)}(x) \geq k$, ne deriva che $f_0^{(k)}(x)+1 \geq k+1$

quindi si ha che $f_0^{(k+1)}(x) \geq k+1$.

<u>LEMMA 3</u>: $\mathbf{f_n(x) > x}$
DIM (per induzione su n): Per $n = 0$ si ha $f_0(x) > x$ per ogni x (per definizione). Assumiamo il risultato valido per $n = k$, cioè che $f_k(x) > x$ per ogni x, e dimostriamo che $f_{k+1}(x) > x$ per ogni x:
- per $x = 0$ si ha $f_{k+1}(0) = f_k^{(0)}(1) = 1 > 0$
- supponiamo che sia vera per $x = m$ e dimostriamolo per $x = m+1$:
  $f_{k+1}(m+1) = f_k(f_{k+1}(m))$ per il lemma 1
  $f_k(f_{k+1}(m)) > f_{k+1}(m)$ per l'ipotesi di induzione su k
  $f_{k+1}(m) > m$ per l'ipotesi di induzione su x
  $f_{k+1}(m) \geq m+1$
  Quindi si ha che $f_{k+1}(m+1) > m+1$.

<u>LEMMA 4</u>: $\mathbf{f_n(x+1) > f_n(x)}$ (funzione crescente)
DIM (per induzione su n): Per $n = 0$ si ha $f_0(x+1) > f_0(x)$ per definizione di f_0. Assumiamo il risultato valido per n e dimostriamo che $f_{n+1}(x+1) > f_{n+1}(x)$:
- $f_{n+1}(x+1) = f_n(f_{n+1}(x))$ per il lemma 1
- $f_n(f_{n+1}(x)) > f_{n+1}(x)$ per il lemma 3

LEMMA 5: $\mathbf{f_{n+1}(x) \geq f_n(x)}$ (successione crescente)

DIM: si ha che:

- $f_{n+1}(x+1) = f_n(f_{n+1}(x))$ per il lemma 1
- Sapendo per il lemma 3 che $f_{n+1}(x) > x$ e quindi $f_{n+1}(x) \geq x+1$, allora
- $f_n(f_{n+1}(x)) \geq f_n(x+1)$ per il lemma 4

LEMMA 6: $\mathbf{f_n^{(k+1)}(x) > f_n^{(k)}(x)}$

DIM: Sappiamo che $f_n^{(k+1)}(x) = f_n(f_n^{(k)}(x))$ e $f_n(f_n^{(k)}(x)) > f_n^{(k)}(x)$ per il lemma 3.

LEMMA 7: $\mathbf{f_n^{(k+1)}(x) \geq 2f_n^{(k)}(x)}$ **(per n≥1)**

DIM (per induzione su k): Per k = 0 si ha:

- $f_n^{(1)}(x) = f_n(x) \geq f_1(x)$ per il lemma 5
- $f_1(x) = 2x$ per definizione di f_1
- $2x = 2f_n^{(0)}(x)$ per definizione di $f^{(0)}$

Quindi $f_n^{(0+1)}(x) \geq 2f_n^{(0)}(x)$.

Assumiamo che il risultato sia vero per k+1 e dimostriamo che $f_n^{(k+2)}(x) \geq 2f_n^{(k+1)}(x)$:

- $f_n^{(k+2)}(x) = f_n^{(k+1)}(f_n(x))$
- $f_n^{(k+1)}(f_n(x)) \geq 2f_n^{(k)}(f_n(x))$ per l'ipotesi di induzione su k
- $2f_n^{(k)}(f_n(x)) = 2f_n^{(k+1)}(x)$

Quindi $f_n^{(k+2)}(x) \geq 2f_n^{(k+1)}(x)$.

LEMMA 8: $\mathbf{f_n^{(k+1)}(x) \geq f_n^{(k)}(x)+x}$ **(per n≥1)**

DIM: Per k = 0 si ha:

- $f_n^{(1)}(x) = f_n(x) \geq f_1(x)$ per il lemma 5
- $f_1(x) = 2x$ per definizione di f_1
- $2x = 2f_n^{(0)}(x)$ per definizione di $f^{(0)}$
- $2f_n^{(0)}(x) = f_n^{(0)}(x) + f_n^{(0)}(x) = f_n^{(0)}(x)+x$

Quindi $f_n^{(0+1)}(x) \geq f_n^{(0)}(x)+x$.

Per k > 0 si ha:

- $f_n^{(k+1)}(x) \geq 2f_n^{(k)}(x)$ per il lemma 7
- $2f_n^{(k)}(x) = f_n^{(k)}(x) + f_n^{(k)}(x)$
- $f_n^{(k)}(x) + f_n^{(k)}(x) \geq f_n^{(k)}(x) + f_n^{(1)}(x)$ per il lemma 6
- $f_n^{(k)}(x) + f_n^{(1)}(x) > f_n^{(k)}(x) + x$ per il lemma 3

Quindi $f_n^{(k+1)}(x) \geq f_n^{(k)}(x)+x$.

LEMMA 9: $\mathbf{f_1^{(k)}(x) \geq (2^k)x}$

DIM (per induzione su k): Per k = 0 si ha $f_1^{(0)}(x) = x = (2^0)x$.

Assumiamo il risultato valido per k e dimostriamo che $f_1^{(k+1)}(x) \geq (2^{k+1})x$:

- $f_1^{(k+1)}(x) = f_1(f_1^{(k)}(x))$
- $f_1(f_1^{(k)}(x)) \geq f_1((2^k)x)$ per il lemma 4 e per l'ipotesi d'induzione su k
- $f_1((2^k)x) \geq 2f_1^{(0)}((2^k)x)$ per il lemma 7
- $2f_1^{(0)}((2^k)x) = 2(2^k)x = (2^{k+1})x$

Proprietà di crescita forte: La funzione $f_n^{(k)}(x)$ è crescente sia in x, sia in n e sia in k.

RIEPILOGO

Introduciamo la seguente notazione:
$g^{(m)}(x) = g(g(\dots g(x)))$ composizione di g con se stessa m volte
$g^{(0)}(x) = x$

Definiamo la seguente famiglia di funzioni:

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

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

questa funzione ci servirà per fornire un limite alla complessità di calcolo di un programma.

Si ha che:
$f_1(x) = 2x$ (con $x \neq 0$)
$f_2(x) = 2^x$
$f_3(x) = 2^{2\text{^}2\text{^} \dots 2 \text{ (x volte)}}$

<u>LEMMA 1</u>: $\mathbf{f_{n+1}(x+1) = f_n(f_{n+1}(x))}$

<u>LEMMA 2</u>: $\mathbf{f_0^{(k)}(x) \geq k}$

<u>LEMMA 3</u>: $\mathbf{f_n(x) > x}$

<u>LEMMA 4</u>: $\mathbf{f_n(x+1) > f_n(x)}$

<u>LEMMA 5</u>: $\mathbf{f_{n+1}(x) \geq f_n(x)}$

<u>LEMMA 6</u>: $\mathbf{f_n^{(k+1)}(x) > f_n^{(k)}(x)}$

<u>LEMMA 7</u>: $\mathbf{f_n^{(k+1)}(x) \geq 2f_n^{(k)}(x)}$ **(per $n \geq 1$)**

<u>LEMMA 8</u>: $\mathbf{f_n^{(k+1)}(x) \geq f_n^{(k)}(x)+x}$ **(per $n \geq 1$)**

<u>LEMMA 9</u>: $\mathbf{f_1^{(k)}(x) \geq (2^k)x}$