LINGUAGGIO DI PROGRAMMAZIONE L
(PROGRAMMI CICLO)

Il linguaggio di programmazione che adesso introdurremo presenta due caratteristiche significative

1) i programmi di L calcolano esattamente le funzioni ricorsive primitive.

2) è possibile introdurre, in modo naturale, una nozione di complessità di calcolo.

Tale linguaggio CICLO (o LOOP) è stato introdotto nel 1968 da A. Meyer e D.M. Ritchie "The Complexity of LOOP Programs", 22ª ACM Nat. Conf. (Washington).

la sintassi di L è del tutto simile a quella del linguaggio $\mathcal{G}$.

la differenza risiede essenzialmente nelle istruzioni base del linguaggio.

Le istruzioni base di L sono

1. $V \leftarrow 0$
2. $V \leftarrow V+1$
3. $V \leftarrow V'$
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 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 $V \leftarrow 0$, $V \leftarrow V+1$, $V \leftarrow V'$ non possono farlo, possiamo concludere che il linguaggio di programmazione L, a differenza del linguaggio S, non dà la possibilità di scrivere programmi che non terminano.
Tutti i programmi di L prima o poi si fermano.
Quindi se un programma, inutilmente complicato, di L non si è fermato possiamo essere sicuri che non ha finito di calcolare e che dobbiamo ancora attendere. (non inutilmente). (L)

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 "cicla 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 il prodotto.

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

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 <u>non</u> 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 <u>non</u> 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 sua 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 $\mathcal{P}_n$ la corrispondente 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 sotto le operazioni di composizione e ricorsione.

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

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

```
Z ← X
Z ← Z + 1
Y ← Z
```

```
Y ← X + 1   Z ← X
Y ← X       Y ← 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 ciclo-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 m 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.

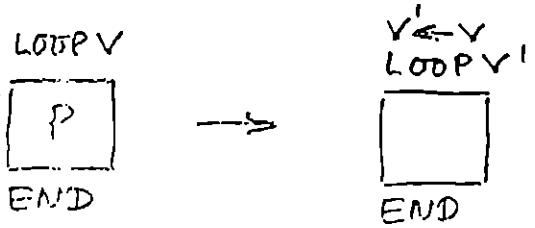
Il nucleo di tale dimostrazione risiede nel mostrare che i meccanismi (algoritmici) di trasformazione dei valori delle variabili che si possono mettere in atto mediante le istruzioni del linguaggio LOOP sono tutti "simulabili" mediante meccanismi esprimibili all'interno della ricorsività primitiva.

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 un marchingegno 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:

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

Immaginiamo adesso di considerare il programma P come il blocco interno da eidare 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 $z_1, z_2, \ldots, z_n, V$:

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

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:

OSS Per calcolare g_i al passo presente devo prima calcolare $g_1, g_2, \dots, g_n$ al passo precedente che non sono ricorsive primitive

la codifica mantiene la ricorsività

Calcolo $\tilde{g}$ al passo presente come funzione delle f_i (che sono ricorsive primitive) con parametri la $\tilde{g}$ al passo precedente (MECCANISMO DI RICORSIONE (L13))

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 sono ricorsive primitive perché la ricorsione non si presenta nella forma semplice che sappiamo preserva la ricorsività primitiva.

Utilizziamo allora dei meccanismi di codifica.

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

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 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 la $\tilde{g}$ dalla $(*)$?

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

Poiché si ha che

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

la proposizione è dimostrata.

PROPOSIZIONE 2

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

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

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

Allora le funzioni $f_1,\dots,f_n$ 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:

- $f_i(z_1, \ldots, z_n) = z_j + k$
- $f_i(z_1, \ldots, z_n) = k$, 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}.

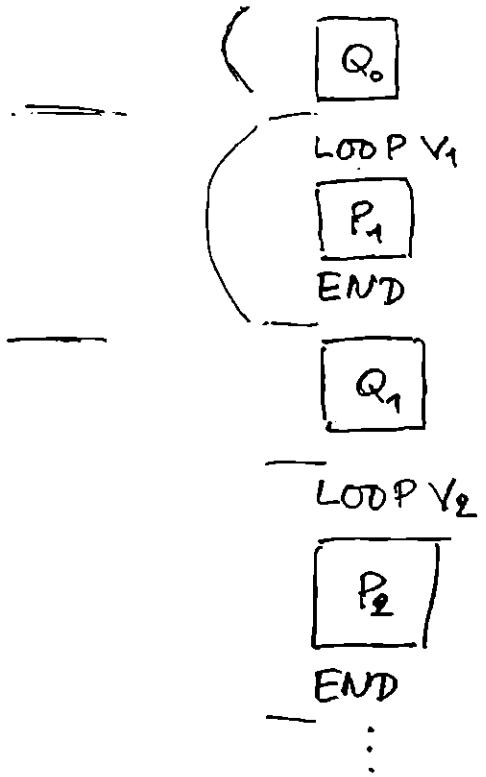
Un programma di L_{n+1} si può decomporre in una serie di blocchi successivi ciascuno dei quali forma un programma appartenente ad L_n, eventualmente inseriti in un ciclo LOOP-END Indichiamo con le lettere P_i questi ultimi e con le lettere Q_i gli altri.

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 Y_i
P_i
END

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


Q_0
LOOP V_1
P_1
END
Q_1
LOOP V_2
P_2
END

Abbiamo adesso tutti gli elementi per dimostrare che i programmi-while 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

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

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 nidificazione non maggiore di P.
In simboli: Se $P \in L_n$ allora $T_P \in \mathcal{L}_n$

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 nidificazione di P.

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

Ricordiamo la notazione:

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

(composizione di g con se stessa n volte)

Definiamo adesso la seguente famiglia di funzioni:

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

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

Si ha che:

$$f_1(x) = 2x \quad (\text{con } x \neq 0)$$

$$f_2(x) = 2^x$$

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

Vogliamo studiare alcune proprietà di questa famiglia di funzioni.

Lemma 1

$$f_{n+1}(x+1) = f_n\left(f_{n+1}(x)\right)$$

Ricordiamo che: $f_{n+1}^{(x)}(1) = f_n^{(x)}(1)$

Dimostrazione

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

Lemma 2

$$f_0^{(k)}(x) \geq k$$

$f_0 = \begin{cases} x+1 & \text{per } x=0 \\ x & \text{per } x=1 \\ x+2 & \text{altrimenti.} \end{cases}$

Dimostrazione

Per induzione su k.

- per $k=0$ si ha $f_0^{(0)}(x) = x \geq 0$
- assumiamo adesso il risultato valido per k, vogliamo dimostrare che è valido per $k+1$.

(1) $$f_0^{(k+1)}(x) = f_0\left(f_0^{(k)}(x)\right)$$

poiché $f_0(x) \geq x+1$ per ogni x (per def. di f_0) si ha che:

(2) $$f_0\left(f_0^{(k)}(x)\right) \geq f_0^{(k)}(x) + 1$$ (assumendo come variabile di f_0, $f_0^{(k)}(x)$)

Poiché per l'ipotesi di induzione $f_0^{(k)}(x) \geq k$, ne discende che:

(3) $$f_0^{(k)}(x) + 1 \geq k+1$$ e, quindi, collegando (1), (2) e (3):

$$f_0^{(k+1)}(x) \geq k+1$$

Lemma 3

$f_n(x) > x$

Dimostrazione

per induzione su n.

- per $n=0$, $f_0(x) \left(= \begin{cases} x+1 \\ x+2 \end{cases}\right) > x$ per ogni x.

- assumiamo ora il risultato vero per $n=k$, cioè che $f_k(x) > x$, per ogni x

  mostriamo che $f_{k+1}(x) > x$ per ogni x

  per $x=0$

  $$f_{k+1}(0) = f_k^{(0)}(1) = 1 > 0$$

  assumiamo ora che sia vera per $x=m$

  $$f_{k+1}(m+1) = f_k(f_{k+1}(m)) \quad \text{(per il lemma 1)}$$
  $$> f_{k+1}(m) \quad \text{(per l'ip. di induz. su } k\text{)}$$
  $$\left(\begin{matrix} > m \\ \geq m+1 \end{matrix}\right) \quad \text{(per l'ip. di induzione su } m\text{)}$$

  Poiché $f_{k+1}(m)$ ed m sono interi e nei due ultimi passaggi è stata trovata una disuguaglianza stretta si ha che

  $$f_{k+1}(m+1) > m+1$$

Lemma 4 ✓ funzione crescente

$f_n(x+1) > f_n(x)$

Dimostrazione.

- per $n=0$ discende immediatamente dalla definizione:
  $f_0(x+1) > f_0(x) \quad \binom{(x+1)+1}{(x+2)+1} > \binom{x+1}{x+2}$
- per induzione su n
  abbiamo che:
  $f_n(x+1) = f_{n-1}(f_n(x)) > f_n(x)$
  ↑ Lemma 1 ↑ Lemma 3

Lemma 5 successione crescente

$f_{n+1}(x) \geq f_n(x)$

Dimostrazione

Per il Lemma 3 si ha che $f_{n+1}(x) > x$
e quindi $\boxed{f_{n+1}(x) \geq x+1}$

Il Lemma 4 ci dice che $\boxed{f_n(x+1) > f_n(x)}$

Si ha che

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

↑ Lemma 1 ↑ Lemma 1 (usando presente che $f_{n+1}(x) \geq x+1$ per il Lemma 3)

Lemma 6

$$f_n^{(k+1)}(x) > f_n^{(k)}(x)$$

Dimostrazione

$$f_n^{(k+1)}(x) = f_n\left(f_n^{(k)}(x)\right) > f_n^{(k)}(x)$$

↑ lemma 3

LA FUNZIONE $f_n^{(k)}(x)$ È "CRESCENTE" SIA IN x SIA IN n SIA IN k PROPRIETÀ DI CRESCITA FORTE

Lemma 7

$$f_n^{(k+1)}(x) \geq 2 f_n^{(k)}(x) \quad (\text{per } n \geq 1)$$

Dimostrazione

- per $k = 0$

$$f_n^{(1)}(x) = f_n(x) \geq f_1(x) = 2x = 2 f_n^{(0)}(x)$$

↑ lemma 5 ↑ Def. f_1 ↑ Def. dif$^{(0)}$

- Assumiamo adesso che il risultato sia vero per $k+1$:

$$f_n^{(k+2)}(x) = f_n^{(k+1)}\left(f_n(x)\right) \geq 2 f_n^{(k)}\left(f_n(x)\right) = 2 f_n^{(k+1)}(x)$$

↑ ip. di induz.

Lemma 8

$f_n^{(k+1)}(x) \geq f_n^{(k)}(x) + x \quad$ per $n \geq 1$

Dimostrazione

- per $k=0$ si ha che:

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

- per $k \geq 0$ si ha:

$$\begin{aligned} f_n^{(k+1)}(x) \geq 2 f_n^{(k)}(x) &= f_n^{(k)}(x) + f_n^{(k)}(x) \\ &> f_n^{(k)}(x) + f_n^{(1)}(x) \quad \updownarrow \text{(lemma 6)} \\ &> f_n^{(k)}(x) + x \quad \updownarrow \text{(lemma 3)} \end{aligned}$$

Lemma 9

$f_1^{(k)}(x) \geq 2^k \cdot x$

Dimostrazione

- per $k=0$

$$f_1^{(0)}(x) = x = 2^0 \cdot x$$

- assumiamo il risultato valido per k.

$$f_1^{(k+1)}(x) = f_1(f_1^{(k)}(x)) \geq f_1(2^k \cdot x) \geq 2 f_1^{(0)}(2^k \cdot x) = 2 \cdot 2^k \cdot x = 2^{k+1} \cdot x$$

($\geq$: lemma 4 (dim. per ind.) e ip. induz.; $\geq$: lemma 7.)

Schema dei Lemmi

Lemma 1

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

con:

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

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

Lemma 2

$$f_0^{(k)}(x) \geq k$$

Lemma 3

$$f_n(x) > x$$

Lemma 4

$$f_n(x+1) > f_n(x)$$

Lemma 5.

$$f_{n+1}(x) \geq f_n(x)$$

Lemma 6

$$f_n^{(k+1)}(x) > f_n^{(k)}(x)$$

Lemma 7

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

Lemma 8

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

Lemma 9

$$f_1^{(k)}(x) \geq 2^k \cdot x$$

PROPRIETÀ DI LIMITAZIONE ALLA CRESCITA

Commento

Sia u il valore massimo delle variabili in ingresso del programma P e supponiamo che si sappia che $T_P(x_1, \dots, x_m) \leq f_n^{(k)}(u)$.

Vogliamo calcolare un limite alla crescita dei valori delle variabili.
Cosa può succedere ai valori delle variabili ad ogni passo del programma?

- Essere azzerati (e questo non li fa aumentare);
- avere assegnato il valore di un'altra variabile. (Questo può fare aumentare di molto il valore della singola variabile ma non aumenta il valore max delle variabili);
- essere incrementati di una unità.

Il caso più sfavorevole riguardo all'aumento massimo del valore delle variabili è proprio dato da un aumento di una unità, ripetuto, della variabile che ha il valore massimo.
Quindi se è $T_P(x_1, \dots, x_m) \leq f_n^{(k)}(u)$,
nella situazione più sfavorevole di un aumento di 1 unità, ad ogni passo, della variabile che ha il valore massimo u, il nuovo valore massimo u' sarà:

$$u' \leq u + T_P(x_1, \dots, x_m) \leq u + f_n^{(k)}(u) \leq f_n^{(k+1)}(u)$$

(la disuguaglianza $\leq$ è giustificata dal Lemma 8)

TEOREMA (DELLA LIMITAZIONE)

Sia $P \in L_n$ allora esiste un k tale che

$$T_P(x_1, \ldots, x_m) \leq f_n^{(k)}(\max(x_1, \ldots, x_m))$$

Dimostrazione

(poniamo $u = \max(x_1, \ldots, x_m)$)

per induzione:

- per $n=0$, il programma P non ha cicli e quindi $T_P(x_1, \ldots, x_m) = k$ (numero di istruzioni del programma) ma $k \leq f_0^{(k)}(u)$ (per il lemma 2).
- assumiamo adesso che il risultato sia vero per $n-1$ e sia P un generico programma di L_n.

Procediamo per gradi.

- P, appartenente ad L_n, appartenga anche ad L_{n-1} allora avremo che:

$$T_P(x_1, \ldots, x_m) \leq f_{n-1}^{(k)}(u) \leq f_n^{(k)}(u)$$

($\leq f_{n-1}^{(k)}(u)$: ipot. induz.; $\leq f_n^{(k)}(u)$: Lemma 5)

- Sia adesso P il programma seguente

```
LOOP V
[ Q ]
END
```

con $Q \in L_{n-1}$

cioè: vi è un solo ciclo esterno che porta ad n la profondità di nidificazione.

è conveniente assumere $n > 1$, per cui dobbiamo considerare separatamente ancora un caso, quello in cui $n = 1$ e quindi $Q \in L_0$.
In questo caso sarà:

$$T_Q = q \quad \text{, per cui sarà}$$

$$T_P(x_1, \dots, x_m) = qv \cdot (\text{dove } v \text{ è il valore di } V \text{ in LOOP } V)$$

ma ricordando si avrà

$$qv \leq qu \leq 2^k \cdot u \leq f_1^{(k)}(u)$$

qv: lungh. di Q "eseguita" v volte; $qu \leq 2^k \cdot u$: per un k opportuno; $2^k \cdot u \leq f_1^{(k)}(u)$: Lemma 9

Assumendo adesso $n > 1$, per l'ipotesi di induzione si ha:

(*) $\quad T_Q(x_1, \dots, x_m) \leq f_{n-1}^{(j)}(u)$ per qualche j

Un calcolo di P è ottenuto facendo girare Q per v volte.
<u>DOBBIAMO PERÒ TENER CONTO DEL FATTO CHE AD OGNI NUOVO CICLO I VALORI DELLE VARIABILI POSSONO ESSERE AUMENTATI.</u>
Dobbiamo perciò trovare valori che limitino la crescita possibile delle variabili dopo ogni esecuzione di Q perché nella (*) la limitazione a T_Q fornita dalla $f^{(j)}$ per un j opportuno è funzione del massimo valore delle variabili d'ingresso.

ci viene incontro la proprietà della limitatezza alla crescita dei valori delle variabili che nel nostro caso diviene:

$$T_Q(x_1,\dots,x_m) \le f_{n-1}^{(j)}(u) \Rightarrow$$

$$u' \le u + f_{n-1}^{(j)}(u) \le f_{n-1}^{(j+1)}(u)$$

dove u' è il nuovo valore massimo delle variabili.

OSSERVAZIONE

Il valore $f_n^{(j+1)}(u)$ non è una limitazione trovata una volta per tutte ma sotto l'ipotesi che T sia limitato da $f_{n-1}^{(j)}(u)$. Possiamo applicarla nella nostra dimostrazione quindi solo perché, per l'ipotesi di induzione abbiamo assunto che $T_Q \le f_{n-1}^{(j)}(u)$.

Facendo girare due volte il programma Q avremo che il tempo di calcolo sarà dato dalla somma del tempo necessario a far girare Q con le variabili d'ingresso tali che $\max\{x_1,\dots,x_n\} = u$ (che è dato da $f_{n-1}^{(j)}(u)$) e del tempo necessario a far girare Q quando le variabili di ingresso sono state modificate dalla precedente esecuzione di Q, tempo che è quindi limitato da $f_{n-1}^{(j)}\left(f_{n-1}^{(j+1)}(u)\right)$.

Il tempo di calcolo totale è quindi limitato da:

$$f_{n-1}^{(j)}(u) + f_{n-1}^{(j)}\left(f_{n-1}^{(j+1)}(u)\right) = f_{n-1}^{(j)}(u) + f_{n-1}^{(j+1)}\left(f_{n-1}^{(j)}(u)\right) \leq$$

↑ Lemma 8 considerando $f_{n-1}^{(j)}(u)$ come var.

$$\leq f_{n-1}^{(j+2)}\left(f_{n-1}^{(j)}(u)\right) = f_{n-1}^{(2j+2)}(u)$$

Il valore massimo delle variabili d'ingresso sarà a questo punto limitato da $f_{n-1}^{(2j+3)}(u)$

Abbiamo che sia le variabili che il tempo di calcolo sono limitate:

dopo 1 giro di Q da: $f_{n-1}^{j+1}(u)$

dopo l giri di Q da: $f_{n-1}^{lj+3}(u)$

Dobbiamo quindi aggiungere all'esponente di f un numero pari a $j+2$ per volta.

Un buon limite (per eccesso) per entrambi T e le variabili dopo u esecuzioni di Q è dato da:

$$f_{n-1}^{(u\cdot(j+2))}(u)$$

(L'istruzione LOOP ci dice di eseguire Q v volte, noi non conosciamo v ma sappiamo che è minore o uguale ad u.)

Allora abbiamo che:

$$T_P(x_1,\dots,x_m) \leq T_{Q[u]}(x_1,\dots,x_m) \leq f_{n-1}^{(u\cdot(j+2))}(u) \leq$$

$$\leq f_{n-1}^{(u(j+2))}(f_n(u)) = f_{n-1}^{(u(j+2))}\left(f_{n-1}^{(u)}(1)\right) =$$

(Lemmi 3 e 4; p. Def di $f_n(u)$)

$$= f_{n-1}^{(u(j+2)+u)}(1) = f_{n-1}^{(u(j+3))}(1) \leq f_{n-1}^{(u\cdot 2^{(j+2)})}(1) \leq$$

(Lemma 6)

$$\leq f_{n-1}^{(f_1^{(j+2)}(u))}(1) \leq f_{n-1}^{(f_n^{(j+2)}(u))}(1) = f_n\left(f_n^{(j+2)}(u)\right) =$$

(Lemma 9; Lemmi 5 e 6; Def f_n)

$$= f_n^{(j+3)}(u) = f_n^{(k)}(u) \qquad \text{posto } k = j+3.$$

Abbiamo così dimostrato il teorema per i programmi P della forma:

LOOP
Q
END

Ci rimane da esaminare il caso più generale...

Decomponiamo adesso il nostro programma in pezzi: ciascun pezzo sarà o un sottoprogramma che appartiene ad L_{n-1} oppure un sottoprogramma della forma ovviamente prima $\begin{array}{|l|}\hline \text{LOOP } Q \\ \hline \text{END} \\ \hline \end{array}$ con $Q \in L_{n-1}$.

Le variabili di uscita di ciascun sottoprogramma saranno le variabili di ingresso del sottoprogramma successivo.

Avremo quindi che:

$$T_P(x_1, \dots, x_m) \leq$$

$$\leq f_n^{(k_1)}(x) + f_n^{(k_2)}\left(f_n^{(k_1)}(x)\right) + f_n^{(k_3)}\left(f_n^{(k_2)}\left(f_n^{(k_1)}(x)\right)\right) + \dots$$

$$+ \dots f_n^{(k_s)}\left(\dots\left(f_n^{(k_1)}(x)\right)\right) \leq f_n^{k}(x)$$

per un opportuno k

applicando il lemma 8 dopo aver osservato che il primo termine e' l'argomento del secondo e così via

volendo calcolare esplicitamente il k osserviamo cosa succede nel caso di una somma di tre soli termini.

$$F = \underbrace{f_n^{(k_1)}(u) + f_n^{(k_2)}\left(f_n^{(k_1)}(u)\right)} + f_n^{(k_3)}\left(f_n^{(k_2)}\left(f_n^{(k_1)}(u)\right)\right)$$

la somma dei primi due addendi, ponendo $f_n^{(k_1)}(u) = Z$, diviene

$$Z + f_n^{(k_2)}(Z) \underset{\text{Lemma 8}}{\leq} f_n^{(k_2+1)}(Z) = f_n^{(k_2+1)}\left(f_n^{(k_1)}(u)\right)$$

Allora abbiamo che:

$$F \leq f_n^{(k_2+1)}\left(f_n^{(k_1)}(u)\right) + f_n^{(k_3)}\left(f_n^{(k_2)}\left(f_n^{(k_1)}(u)\right)\right)$$

Adesso non è più vero che il primo termine è l'argomento della funzione $f_n^{(k_3)}$.

Osserviamo però che, per il lemma 4,

$$f_n^{(k_3)}\left(f_n^{(k_2)}\left(f_n^{(k_1)}(u)\right)\right) \leq f_n^{(k_3)}\left(f_n^{(k_2+1)}\left(f_n^{(k_1)}(u)\right)\right)$$

poiché

$$f_n^{(k_2)}\left(f_n^{(k_1)}(u)\right) < f_n^{(k_2+1)}\left(f_n^{(k_1)}(u)\right)$$

per il lemma 6

....

Allora avremo che

$$F < f_n^{(k_3+1)}\left(f_n^{(k_1)}(u)\right) + f_n^{(k_3)}\left(f_n^{(k_2+1)}\left(f_n^{(k_1)}(u)\right)\right)$$

osservando che il primo termine è, adesso, l'argomento di $f_n^{(k_3)}$, possiamo applicare il lemma 8 ed ottenere che

$$F < f_n^{(k_3+1)}\left(f_n^{(k_2+1)}\left(f_n^{(k_1)}(u)\right)\right)$$

$$= f_n^{[(k_3+1)+(k_2+1)+k_1]}(u)$$

Contiamo come compaiono gli "esponenti": sono tutti aumentati di 1 tranne il primo.

3 + 1 + 1

Ci si convince facilmente che il meccanismo resta invariato per una somma di s termini per cui si ha che il costo T_p è maggiorato da:

$$f_n^{[(k_s+1)+(k_{s-1}+1)+(k_{s-2}+1)+\dots+(k_2+1)+k_1]}(u) =$$

$$= f_n^{(k_1+k_2+\dots+k_s+s-1)}(u)$$

31 quater

usando ancora una volta la proprietà di limitazione alla crescita. avremo allora il

COROLLARIO

Se $g \in \mathcal{L}_n$ allora esiste una costante k tale che

$$g(x_1, \dots, x_m) \leq f_m^{(k)}(\mathrm{MAX}(x_1, \dots, x_m)).$$

Fino ad ora ci siamo sempre comportati come se effettivamente $\mathcal{L}_{n-1} \subset \mathcal{L}_n$, cioè $\mathcal{L}_n$ contiene qualche funzione che non è calcolabile mediante programmi di L_{n-1}.

Mostriamo adesso che è effettivamente così.

Proposizione

Per $n \geq 1$, $f_n \in \mathcal{L}_n$

Dimostrazione

per induzione su n

[redacted]

— per $n = 1$ $\quad f_1(0) = 1$

$f_1(x) = 2x$ per $x > 0$

f_1 è calcolata dal programma:

```
Y ← Y+1   ← (questa per avere in uscita 1 quando X è zero)
LOOP X
   X ← X+1
   Y ← X
END
```

- Scriviamo adesso un programma che calcoli f_{k+1}. Per definizione $f_{k+1}(x) = f_k^{(x)}(1)$ quindi

```
Y ← Y+1
Z ← Z+1
LOOP X
   Y ← fk(Z)
   Z ← Y
END
```

Per l'ipotesi di induzione $f_k \in \mathcal{L}_k$ e quindi $f_{k+1} \in \mathcal{L}_{k+1}$, perché abbiamo appunto un nuovo ciclo... L'ipotesi di induzione $f_k \in \mathcal{L}_k$ ci garantisce anche che possiamo scrivere la macro $Y \leftarrow f_k(Z)$ (perché f_k è ricorsiva primitiva)

Per ottenere il nostro risultato ci resta da mostrare che $f_n \notin \mathcal{L}_{n-1}$.
Useremo sempre dei meccanismi di maggiorazione.
Per fare ciò avremo bisogno della nozione di una funzione che è maggiore di un'altra almeno da "un certo punto in poi".
Ciò può formalizzarsi con la nozione di predicato vero quasi ovunque, cioè vero tranne che su un insieme finito di numeri.

Allora diremo che f è una maggiorante di g ($f \succ g$) se $f(x) > g(x)$ quasi ovunque.

Proposizione:

$f_{n+1} \succ f_n^{(k)}$ per ogni n e k.

Dimostrazione

per $n=0$ $\quad f_0^{(k)}(x) = x + \underbrace{2 + \ldots + 2}_{k \text{ volte}} = x + 2k \quad (\text{per } x \geq 2)$

$f_1(x) = 2x$

e quindi è vera perché $2x \succ x + 2k$
essendo $x > 2k$, per ogni k fissato, da un certo punto in poi.

Sia adesso $n \neq 0$ e consideriamo per induzione su k

per $k = 0$ si ha che:

$$f_{n+1}(x) > x = f_n^{(0)}(x)$$

($f_{n+1}(x) > x$: lemma 3; $x = f_n^{(0)}(x)$: Def. $f^{(0)}$)

Poniamo quindi supporre $f_{n+1} > f_n^{(k)}$ e vogliamo mostrare che è anche $f_{n+1} > f_n^{(k+1)}$.

Allora avremo che (da un certo punto in poi):

$$f_n^{(k+1)}(x) < f_n^{(k+1)}(2n-4) = f_n^{(k+1)}(f_1(n-2)) \leq$$

($<$: Lemma 4 (da un certo punto in poi); $=$: per def. di f_1)

$$\leq f_n^{(k+1)}(f_n(n-2)) = f_n^{(2)}(f_n^{(k)}(n-2)) \leq$$

($\leq$: Lemma 4 e 5)

$$\leq f_n^{(2)}(f_{n+1}(n-2)) = f_n^{(2)}(f_n^{(n-2)}(1)) = f_n^{(n)}(1) =$$

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

($\leq$: ipotesi di induzione (da un certo punto in poi))

cioè:

$$f_n^{(k+1)}(n) < f_{n+1}(n) \quad \text{quasi ovunque.}$$

Possiamo adesso dimostrare la

PROPOSIZIONE

$f_{n+1} \in \mathcal{L}_{n+1}$ ma $f_{n+1} \notin \mathcal{L}_n$

Dimostrazione

Sappiamo che $f_{n+1} \in \mathcal{L}_{n+1}$

Supponiamo che sia $f_{n+1} \in \mathcal{L}_n$

Per il corollario del teorema della limitazione si ha che (essendo $f_{n+1}(x)$ supposta $\in \mathcal{L}_n$)

$$f_{n+1}(x) \leq f_n^{(k)}(x)$$ per qualche k e per quasi x

ma per la proposizione appena mostrata si ha che:

$$f_n^{(k)}(x) < f_{n+1}(x)$$ quasi ovunque

allora avremmo che

$$f_{n+1}(x) < f_{n+1}(x)$$ quasi ovunque.

Contraddizione –

Non possiamo quindi assumere che $f_{n+1} \in \mathcal{L}_n$.

Osservazione

Notiamo che i risultati precedenti mostrano due cose in un colpo solo:

- che $\mathcal{L}_n$ è una gerarchia e che quindi aumentando la profondità di ridificazione riusciamo a calcolare effettivamente più cose.
- che tutte le f_n sono ricorsive primitive. (cosa che prima potevamo sospettare ma che non avevamo ancora dimostrato).

Mostreremo adesso l'esistenza di una funzione calcolabile ma non ricorsiva primitiva.
Questo risultato lo avevamo già preannunciato, adesso possiamo anche dimostrarlo avendo a disposizione tutto l'apparato formale necessario.
Si osservi che il termine calcolabile è da intendersi sia in senso intuitivo che in senso formale (in particolare lo intenderemo nel senso di calcolabile in $\mathcal{S}$).

La funzione che andremo adesso ad esaminare è la funzione di Ackermann di due variabili definita da:

$$A(i,x) = f_i(x)$$

che gode delle proprietà:

$$\begin{cases} A(i+1,x+1) = A(i, A(i+1,x)) \\ A(i,0) = 1 \\ A(0,x) = \begin{cases} x+1 & \text{se } x = 0,1 \\ x+2 & \text{se } x > 1 \end{cases} \end{cases}$$

TEOREMA

La funzione $A(x,x) = f_x(x)$ non è ricorsiva primitiva

Dimostrazione

Poniamo $A(x,x) = f_x(x) = g(x)$

Supponiamo che g sia ricorsiva primitiva.

Allora, poiché sappiamo già che le funzioni ricorsive primitive sono calcolabili da programmi-ciclo, esisterà un m tale che:

$$g \in \mathcal{L}_m$$

ma, in base ai teoremi sulla limitazione alla ricorsione,

$$g \in \mathcal{L}_m \Rightarrow \exists k \text{ tale che } g(x) \leq f_m^{(k)}(x)$$

sappiamo ancora che:

$$f_m^{(k)}(x) < f_{m+1}(x) \quad \underline{\text{quasi ovunque}}$$

pertanto sarà

(X) $$g(x) < f_{m+1}(x)$$ quasi ovunque (cioè da un certo valore di x in poi)

Sia n_0 un intero $> m+1$ per cui la disuguaglianza è verificata, allora avremo che:

$$f_{n_0}(n_0) = g(n_0) < f_{m+1}(n_0)$$

D'altronde sappiamo che $f_p(x) < f_q(x)$ per $p < q$ e quindi, essendo $n_0 > m+1$, sarà:

$$f_{n_0}(n_0) > f_{m+1}(n_0)$$ CONTRADDIZIONE.

Abbiamo con- dimostrato che $A(x,x)$ non è- ricorsiva primitiva.

Ora possiamo dire di $A(x,y)$?

Possiamo immediatamente concludere che anche $A(x,y)$ non è- ricorsiva primitiva, perché se lo fosse stata allora anche $A(x,x)$ sarebbe stata ricorsiva primitiva.

Presentiamo adesso un procedimento per calcolare $A(i,x)$ che ben si presta ad essere programmato (Ricordiamo che, in passato, abbiamo usato il procedimento di calcolo di A per introdurre la nozione di funzione (ε-)ricorsiva.

USANDO UNA PARTICOLARE MEMORIA DETTA "PILA" IN CUI IMMAGAZZINARE GLI ARGOMENTI DI SINISTRA MENTRE STIAMO VALUTANDO QUELLI DI DESTRA.

LA PILA SARÀ INDICATA DA $(a_1, \dots, a_m)$ E SI CONVERRÀ CHE a_1 È L'ELEMENTO PIÙ IN ALTO NELLA PILA E, IN GENERALE, a_i STA PIÙ IN ALTO DI a_{i+1}.

Vogliamo adesso calcolare $A(2,2) = A(1, A(2,1))$; Poniamo allora 1 nella pila e calcoliamo $A(2,1)$:

$A(i,0)=1$ $A(0,x)=\begin{cases} x+1 & \text{se } x=0 \\ x+2 & \text{se } x>1 \end{cases}$		$A(2,2)=A(1, A(2,1))$
	(1)	$A(2,1)=A(1, A(2,0))$
	(1,1)	$A(2,0)=1$
	(1)	$A(1,1)=A(0, A(1,0))$
$A(i+1,x+1)=$ $A(i, A(i+1,x))$	(0,1)	$A(1,0)=1$
	(1)	$A(0,1)=2$
	(—)	$A(1,2)=A(0, A(1,1))$
	(0)	$A(1,1)=A(0, A(1,0))$
	(0,0)	$A(1,0)=1$
	(0)	$A(0,1)=2$
	(—)	$A(0,2)=4$

Il procedimento si può separare nei seguenti passi:

1. Si mette l'argomento di sinistra nella pila e si sviluppa l'espressione a destra iterativamente fino ad arrivare ad espressioni del tipo $A(i,0)$ e $A(0,x)$.
2. Le espressioni $A(i,0)$ e $A(0,x)$ si fanno calcolare immediatamente, per cui, ricorsivamente:
3. Si prende il valore più in alto della pila, e si opera successivamente come sul 1 nella nuova [illegible]

Per avere nel nostro programma questo nuovo oggetto che è la pila abbiamo bisogno di due variabili L ed S la prima delle quali ci informa nella lunghezza della pila stessa e la seconda contiene i valori che abbiamo conservato nella pila.

L'operazione di porre q nella pila è rappresentato da:

$L \leftarrow L+1$ (aumenta di 1 la lunghezza)
$S \leftarrow \langle q, S \rangle$ (codifica mediante la $\langle , \rangle$ il nuovo q)

L'operazione inversa di prendere il primo elemento della pila è invece rappresentata da:

$L \leftarrow L-1$ (diminuisci di 1 la lunghezza)
$i \leftarrow l(S)$ (prendi l'elemento sinistro di $S = \langle , \rangle$ che è quello inserito)
$S \leftarrow r(S)$ (rimetti in S ciò che resta, cioè il lato destro di $\langle , \rangle$)

PROGRAMMA CHE CALCOLA LA FUNZIONE DI ACKERMANN $A(i, n)$

definizione: $A(i, n) = A(i-1, A(i, n-1))$

con $A(i, 0) = 1$

e $A(0, n) = \begin{cases} n+1 & \text{se } n = 0, 1 \\ n+2 & \text{se } n > 1 \end{cases}$

```
1.  [A]  IF X ≠ 0 GOTO D        se n=0 calcola A(i,0) = 1
2.       X ← X - 1
3.  [B]  IF L = 0 GOTO G        Se la pila non è vuota
4.       L ← L - 1              prende il primo valore
5.       i ← ℓ(S)               dalla pila
6.       S ← τ(S)
7.       GOTO A
8.  [C]  X ← X + 2              calcola A(0,n) = n+2
9.       GOTO B                 per n > 1 che è il caso
10. [D]  IF i ≠ 0 GOTO F        in cui viene attivata..
11.      IF X ≠ 1 GOTO C
12.      X ← 2                  calcola A(0,1)
13.      GOTO B
14. [F]  X ← X - 1              calcola in generale
15.      L ← L + 1              A(i,n) = A(i-1, A(i,n-1))
16.      S ← <i-1, S>
17.      GOTO A
18. [G]  Y ← X
```

Sono possibili quattro casi:

1. $A(i,0) = 1$ realizzato dalle istruzioni $1 \to 2 \to 3 \to 18$
2. $A(0,1) = 2$ " da $1 \to 10 \to 11 \to 12 \to 13 \to 3 \to 18$
3. $A(0,x) = x+2$ per $x>1$: $A \to D \to C$ (se non ci trovo nella pila)
   $1 \to 10 \to 11 \to 8 \to 9 \to 3 \to 18$
4. $A(i,x) = A(i-1, A(i,x-1))$ con $x, i \neq 0$ oppure $i=0$ e $x>1$

   ciclo: $A \to D \to F \to A$ che calcola tale valore in generale

   ciclo: $A \to D \to C \to B \to A$ che considera il caso $i=0$ e $x>1$ per cui $A(0,x)=x+2$ e che va a prendere il nuovo valore dalla pila

11.

- IL GRUPPO DI ISTRUZIONI 1-2 calcola $A(i,0)=1$ oppure rimanda al GRUPPO [D] E SUCCESSIVE (CHE POTREMMO CHIAMARE "GRUPPO SMISTAMENTO")
- INFATTI, [D] 10-11-12 SELEZIONA I CASI:
  - $(x=0, i \neq 0) \to$ [F] (calcolo generale)
  - $(x \neq 0, i=0, x \neq 1) \to$ [C] (calcolo $A(0,x)$ per $x>1$)
  - $(x=1, i=0) \to$ (calcolo di $A(0,1)$).
- IL GRUPPO [B] 3-4-5-6 prende il primo valore della pila (infatti viene attivato o dopo il calcolo di $A(0,x)$ o dopo quello di $A(0,1)$)
- IL GRUPPO [F] 14-15-16-17 sviluppa, in generale, $A(i,x)$ e, infatti viene attivato solo nel caso generale ($i \neq 0$, $x \neq 0$)

40bis

Osservazioni

1. L'esibizione del programma di S per calcolare $A(i,n)$ non era necessaria per mostrare che A è calcolabile da un punto di vista intuitivo, perché la definizione stessa mostra in maniera più che sufficiente l'esistenza di un procedimento algoritmico, per quanto complicato, per calcolare i suoi valori.

2. Quindi la dimostrazione dell'insufficienza della ricorsività primitiva a ricoprire il campo della calcolabilità si ha già con la sola dimostrazione della non ricorsività primitiva di $A(i,n)$.

3. La costruzione del programma è invece necessaria per mostrare che $A(i,n)$ è <u>calcolabile</u> in S; cioè è necessaria se si vuole mostrare che S è sufficientemente potente da calcolare anche questa funzione che <u>non</u> è ricorsiva primitiva.

4. Una volta data <u>una</u> dimostrazione di calcolabilità di $A(i,n)$ in <u>un</u> formalismo non c'è ovviamente alcun bisogno di ripetere la dimostrazione in altri formalismi dimostrabilmente equivalenti.

Una caratteristica positiva del linguaggio LOOP è quella di non avere istruzioni di salto.
Vediamo se è possibile arricchirlo rinunciando però a non perdere il potere espressivo di $\mathcal{L}$.
Aggiungiamo alle istruzioni di L la seguente coppia di istruzioni, che funziona in modo simile a quello della coppia LOOP-END:

WHILE V≠0 DO (Questo linguaggio L + WHILE sarà chiamato W)
[- -] - - - - - -
END

Quindi mentre V≠0 il blocco di istruzioni compreso tra WHILE ed END deve essere ripetuto
Si osservi che questa volta, a differenza del caso LOOP-END, il valore di V può variare nel corso del calcolo ed eventualmente non divenire mai zero.
In questo caso, quindi, non abbiamo un controllo a priori sul numero di volte che il blocco di istruzioni compreso tra WHILE ed END verrà eseguito.
Si osservi ancora che

il seguente di programma

WHILE V≠0 DO
[P]
END

può essere simulato da

V=0
[A] IF GOTO B
[P]
GOTO A
[B] ——

Quindi tutte le funzioni calcolabili da programmi di W sono calcolabili in $\mathcal{L}$.

Richiamiamo adesso un teorema nella rappresentazione delle funzioni ricorsive (non ancora dimostrato) che ci permetterà di dimostrare che vale anche la proprietà inversa.

Ogni funzione parzialmente computabile f di n variabili $x_1, \ldots, x_n$ può essere scritta come:

$$(*) \quad f(x_1,\ldots,x_n) = l(\min_z [g(x_1,\ldots,x_n,z)=0])$$

dove g è una funzione ricorsiva primitiva, l una delle funzioni (ricorsive primitive) coppia e $\min_z$ l'operatore di minimalizzazione non limitata.

TEOREMA: Ogni funzione parzialmente calcolabile può essere calcolata da un programma del linguaggio W.

Dimostrazione

La nostra funzione f sia rappresentata come in $(*)$. Allora essa è calcolata dal seguente programma in W:

```
1.    Z ← 0
2.    V ← g(X1,...,Xn,0)
3.    WHILE V ≠ 0 DO
4.        Z ← Z+1
5.        V ← g(X1,...,Xn,Z)
6.    END
7     Y ← l(Z)
```

Osservazioni

1. Le linee 2, 5, 7 sono ammissibili in L (e quindi in W) perché g ed l sono ricorsive primitive.
2. In base al teorema precedente non solo sono sempre evitabili le istruzioni di salto ma è sufficiente usare una sola istruzione WHILE.

zione delle funzioni ricorsive (non ancora dimostrata) che ci permetterà di dimostrare che vale anche la proprietà inversa.

Ogni funzione parzialmente computabile f di n variabili $x_1, \dots, x_n$ può essere scritta come:

$$(*) \quad f(x_1,\dots,x_n) = \ell(\min_z [g(x_1,\dots,x_n,z)=0])$$

dove g è una funzione ricorsiva primitiva, ℓ una delle funzioni (ricorsive primitive) copia e $\min_z$ l'operatore di minimalizzazione non limitata.

TEOREMA: Ogni funzione parzialmente calcolabile può essere calcolata da un programma del linguaggio W.

Dimostrazione

La nostra funzione f ha rappresentata come in (*). Allora essa è calcolata dal seguente programma in W:

```
1.    Z ← 0
2.    V ← g(X1,...,Xn,0)
3.    WHILE V ≠ 0 DO
4.        Z ← Z+1
5.        V ← g(X1,...,Xn,Z)
6.    END
7     Y ← ℓ(Z)
```

Osservazioni

1. Le macro 2,5,7 sono ammissibili in L (e quindi in W) perché g ed ℓ sono ricorsive primitive.
2. In base al teorema precedente non solo sono sempre evitabili le istruzioni di salto ma è sufficiente usare una sola istruzione WHILE.

Schema dei lemmi

Lemma 1

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

Lemma 2

$$f_0^{(k)}(x) \geq k$$

Lemma 3

$$f_n(x) > x$$

Lemma 4

$$f_n(x+1) > f_n(x)$$

Lemma 5.

$$f_{n+1}(x) \geq f_n(x)$$

Lemma 6

$$f_n^{(k+1)}(x) > f_n^{(k)}(x)$$

Lemma 7

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

Lemma 8

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

Lemma 9

$$f_1^{(k)}(x) \geq 2^k \cdot x$$

cioè

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

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