Il Linguaggio $\mathcal{L}$: Parte I*

VERSIONE PRELIMINARE
Gli studenti sono vivamente invitati a studiare anche sul libro e a comunicare possibili errori e imprecisioni

May 30, 2013

Il linguaggio $\mathcal{L}$, chiamato anche linguaggio CICLO (o LOOP), che adesso esamineremo è stato introdotto nel 1967 da A. Meyer e D. M. Ritchie [**?**]. Lo scopo principale degli autori di questo lavoro era quello di trovare limitazioni al tempo di calcolo di ampie categorie di programmi interessanti che comprendessero le funzioni aritmetiche che si incontrano nella pratica quotidiana. L'indecidibilità del teorema della fermata impedisce di cercare tali limitazioni per TUTTI i programmi di un linguaggio di programmazione sufficientemente generale, ma possiamo farlo per opportune sottoclassi. Una caratteristica estremamente interessante del lavoro di Meyer e Ritchie è che loro studiano tali limitazioni per una sottoclasse estremamente significativa delle funzioni calcolabili, le funzioni ricorsive primitive e lo fanno usando un linguaggio al contempi estremamente semplice e profondo. Aspetti, questi, che rendono il loro lavoro ancora utile ed attuale - dopo più di quarantacinque anni - anche per scopi didattici. Un corollario del loro lavoro è quello di mostrare la perfetta equivalenza delle due nozioni di funzione calcolabile da linguaggi CICLO e di funzione ricorsiva primitiva, fornendo così una visualizzazione utilissima di tipo puramente informatico di questa classe di funzioni che aveva svolto un ruolo importante negli sviluppi puramente teorici durante il percorso che ha condotto a precisare e formalizzare la nozione intuitiva di calcolabile. Ricordiamo che una trattazione completa e dettagliata delle caratteristiche e proprietà delle funzioni ricorsive primitive è dovuta a Rózsa Péter [**?**]. La trattazione che qui sarà seguita è quella del capitolo 13 del libro di Davis e Weyuker [**?**]

1 Introduzione al linguaggio $\mathcal{L}$.

Il linguaggio di programmazione che adesso introdurremo presenta due caratteristiche significative:

1. i programmi di $\mathcal{L}$ calcolano esattamente le funzioni ricorsive primitive;
2. è possibile introdurre, in modo naturale, una nozione di complessità di calcolo.

*Appunti dalle lezioni di Informatica teorica (modulo Teoria della calcolabilità) laurea triennale in Informatica Università degli Studi di Palermo (a. a. 2012/13)

La sintassi di $\mathcal{L}$ è del tutto simile a quella del linguaggio $\mathcal{S}$. La differenza risiede essenzialmente nelle istruzioni base del linguaggio. Le istruzioni 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 deve essere usata forse sarà più chiaro dopo avere presentato alcuni esempi. Adesso ci limitiamo ad affermare che esse fanno ripetere (o "ciclare", da loop = ciclo), ciò che è contenuto tra l'istruzione LOOP e l'istruzione END un numero di volte pari al valore della variabile presente nell'istruzione LOOP al momento in cui questa viene attivata. Ciò fatto, verrà poi eseguita l'istruzione che segue END.

Osservazione Come già scritto, il numero di volte che il blocco di istruzioni contenuto tra LOOP e END deve essere eseguito è dato dal valore della variabile V presente nell'istruzione LOOP nel momento in cui questa è incontrata. Quindi anche se il valore di X viene modificato nel corso dell'esecuzione del ciclo, ciò non ha alcuna influenza sul numero di volte che il blocco di istruzioni compreso tra LOOP ed END deve essere eseguito.

Una conseguenza di ciò è che le istruzioni LOOP-END non possono indurre meccanismi di non terminazione. Poiché anche le altre istruzioni del linguaggio $\mathcal{L}$: V$\leftarrow$ 0, V$\leftarrow$ V+1, V$\leftarrow$ V$'$ non possono farlo, possiamo concludere che $\mathcal{L}$, a differenza del linguaggio $\mathcal{S}$, non permette di scrivere programmi che non terminano. Tutti i programmi di $\mathcal{L}$ prima o poi si fermeranno. Quindi se un programma di $\mathcal{L}$ non si è fermato possiamo essere sicuri che non ha finito di calcolare e che dobbiamo ancora attendere (non inutilmente) per avere il risultato del computo.

Esempio

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 variabile X). La 4 viene perciò eseguita una sola volta ed il valore finale di X è, quindi, 2 (indipendentemente dal valore di ingresso di X perché l'istruzione 1 ha provveduto ad azzerarlo). Il valore 2 di X viene assegnato, infine, mediante l'istruzione 6, alla variabile di uscita Y.

Consideriamo adesso un altro programma di $\mathcal{L}$

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

L'istruzione 2 ci dice di ripetere X_1 volte quello che è compreso fra 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 *nidificazione*.

Possiamo adesso 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 altre istruzioni LOOP-END.

In generale diremo che un programma ha profondità di nidificazione n se vi è almeno una coppia LOOP-END che abbia profondità di nidificazione n, tale cioè che il blocco di istruzioni da essa abbracciato contiene, a sua volta (al suo interno) almeno una coppia nidificata $n-1$ volte, e se al contempo non esiste alcuna coppia LOOP-END con profondità di nidificazione $n+1$. I programmi con profondità di nidificazione 0 sono quelli che *non* contengono istruzioni LOOP-END. I programmi visti prima hanno profondità di nidificazione 1 e 2 rispettivamente.

Poniamoci, adesso, la domanda che segue: "In che modo può essere utilizzata tale nozione di profondità di nidificazione?" Solamente per indicare in modo sintetico come si presenta "graficamente" il programma, oppure ci permette di indicare qualcosa di più significativo? Detto in altro modo, quello che ci chiediamo è se la profondità di nidificazione sia solo una caratteristica *descrittiva* dei programmi di $\mathcal{L}$ oppure

indichi qualcosa di *strutturale*. Vedremo in seguito che essa mette in evidenza importanti proprietà strutturali dei programmi di $\mathcal{L}$ e che può essere considerata come una sorta di misura di complessità. Adesso usiamola come una guida euristica per porre delle domande elementari che ci condurranno in modo naturale ad esaminare caratteristiche significative di questa famiglia di programmi. Riprendiamo in esame il secondo esempio nel quale abbiamo presentato un programma che calcola la funzione *moltiplicazione* (e che ha profondità di nidificazione 2). Quale potrebbe essere una sua *semplificazione* che permetta di calcolare la funzione *somma*? Un esempio è dato dal programma seguente che ha profondità di nidificazione uguale a 1.

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

In modo assolutamente semplice e naturale siamo stati indotti a scrivere un programma che calcola la somma usando un solo ciclo LOOP-END e un programma che calcola il prodotto con due cicli nidificati. Si ha la sensazione che, aumentando la profondità di nidificazione, si possano fare, in modo naturale, cose via via più complesse. Questa è solo una sensazione oppure corrisponde a caratteristiche profonde e strutturali di questo linguaggio? Per rimanere nel concreto, chiediamoci se sia possibile scrivere un programma che calcola il prodotto e che ha profondità di nidificazione 1. Ovviamente tale programma potrebbe non avere la semplicità e l'immediatezza di quello dell'esempio 1 e probabilmente sarebbe molto piu lungo. Ma la domanda riguarda proprio (e solo) la *possibilità* di farlo, non la semplicità e la "leggibilità" del risultato ottenuto. Un'altra domanda che sorge in modo naturale è se abbiamo realmente bisogno di una profondità di nidificazione grande a piacere (non superiormente limitata) per calcolare tutte le funzioni calcolabili da programmi di $\mathcal{L}$. Cioè, detto in altro modo, non ci stupirebbe scoprire che con profondita di nidificazione 2 possiamo fare più cose che con profondita di nidificazione 1 (indipendentemente dall'esempio specifico della domanda precedente riguardante la funzione prodotto) e neanche che con profondità di nidificazione 3 possiamo fare più cose che con profondità di nidificazione 2. Ma il potere usare programmmi con profondità di nidificazione 137.731, tanto per portare un esempio, ci permette realmente di potere fare più cose di quelle che potremmo fare nel caso in cui sono abilitato a usare solo programmi che al massimo hanno profondita di nidificazione 137.730? Renderemo successivamente precise tali considerazioni in modo da rispondere in modo rigoroso a tutte queste questioni poste e troveremo che al crescere della profondità di nidificazione potremo calcolare sempre nuove funzioni e a ogni passo le nuove funzioni sono più complesse di quelle calcolate prima nello stesso senso in cui il prodotto e più complesso della somma.

Adesso procediamo con ordine e introduciamo una notazione che ci sarà utile nel prossimo paragrafo :

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

In base alla terminologia introdotta, L_0 è la classe dei programmi che non contengono istruzioni LOOP-END, e i programmi che abbiamo introdotto prima appartengono, rispettivamente, ad L_1 (somma e funzione costante 2), e ad L_2 (moltiplicazione).

2 Equivalenza tra le funzioni ricorsive primitive e quelle calcolate dal linguaggio $\mathcal{L}$.

In questo paragrafo dimostreremo - come già preannunciato - che i programmi CICLO calcolano esattamente le funzioni ricorsive primitive, quindi la classe delle funzioni ricorsive primitive e la classe delle funzioni calcolate da programmi CICLO sono *estensionalmente* equivalenti.

Proposizione 1 *Le funzioni ricorsive primitive appartengono alla classe $\mathcal{L} = \cup_{n=0}^{\infty}\mathcal{L}_n$ 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 (cioè che possiamo effettuare mediante programmi di $\mathcal{L}$ le operazioni di composizione e ricorsione). Si verifica immediatamente che le funzioni iniziali appartengono ad $\mathcal{L}_0$. Infatti, la funzione successore $s(n)$ è calcolata dal programma:

```
Z ← X
Z← Z+1
Y← Z
```

e le funzioni di selezione $u_i^n(x_1, \ldots, x_n)$ e la funzione costante zero $u(x)$ sono calcolate, rispettivamente, dai programmi seguenti (di una sola istruzione grazie al fatto che tra le istruzioni di L vi è quella di assegnazione):

```
u_i^n(x_1, ..., x_n)
Y ← X_i

u(x)
Y ← 0
```

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 $f(g_1(x_1, \ldots, x_n), \ldots, g_n(x_1, \ldots, x_n))$ dove f e le g_i sono calcolabili da programmi-ciclo è dato da:

$$
\begin{array}{lcl}
Z_1 & \leftarrow & g_1(x_1, \ldots, x_n) \\
\ldots & \ldots & \ldots \\
Z_n & \leftarrow & g_n(x_1, \ldots, x_n) \\
Y & \leftarrow & f(Z_1, \ldots, Z_n)
\end{array}
$$

Ogni riga del programma precedente è, infatti, una macro ammissibile nel nostro linguaggio perché in $\mathcal{L}$ è già disponibile una istruzione di assegnazione e sia la f sia le g_i sono ciclo-calcolabili. La composizione di funzioni in $\mathcal{L}_k$ produce, quindi, una funzione ancora in $\mathcal{L}_k$

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

Ci resta quindi soltanto di mostrare che questo succede anche con l'operazione di ricorsione. Consideriamo direttamente il caso più generale. La funzione h sia definita mediante le funzioni f e g, per ipotesi ciclo-calcolabili, dalle equazioni seguenti.

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

Che la funzione h sia ciclo-calcolabile è mostrato dal programma che segue:

Y $\leftarrow f(x_1, \ldots, x_n)$
Z $\leftarrow 0$
LOOP X_{n+1}
    Y $\leftarrow g(Z, Y, X_1, \ldots, X_n)$
    Z $\leftarrow$ Z+1
END

Osservazione. Se k è la profondità di nidificazione del programma che calcola f ed m è quella del programma che calcola g, allora la profondità di nidificazione del programma che calcola h sarà uguale al valore piu alto tra k ed $m + 1$. Quindi mentre l'operazione di composizione non aumentava la profondita di nidificazione, l'operazione di ricorsione può aumentarla.

Abbiamo così concluso la dimostrazione della proposizione.

Vogliamo mostrare adesso l'inverso della proposizione precedente. A prima vista questa impresa appare disperata perché, se da un lato abbiamo una famiglia di funzioni definita in modo molto chiaro e con regole di composizione semplici, dall'altro lato abbiamo funzioni di cui sappiamo **solo** che sono calcolate da un programma di $\mathcal{L}$ (da un qualsiasi programma!). Dovremmo quindi prendere in considerazione un arbitrario programma di $\mathcal{L}$ e mostrare che esso non può che calcolare una funzione ricorsiva primitiva. Un compito molto arduo se non impossibile. Fortunatamente ci viene in aiuto la particolare struttura del linguaggio $\mathcal{L}$.

Il nucleo della dimostrazione che ogni funzione ciclo-calcolabile è ricorsiva primitiva si basa, infatti, nella possibilità di 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, quindi, le variazioni indotte dall'esecuzione di un programma-ciclo sui valori delle variabili presenti in modo del tutto generale. Per semplificare la visualizzazione di tali trasformazioni faremo, in particolare, 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:

$$\begin{array}{lcl} & & \mathrm{V}' \leftarrow \mathrm{V} \\ \mathrm{LOOP\ V} & & \mathrm{LOOP\ V}' \\ \boxed{\mathrm{P}} & \rightarrow & \boxed{\mathrm{P}} \\ \mathrm{END} & & \mathrm{END} \end{array}$$

Immaginiamo ora un programma $\mathcal{P}$ scritto nel linguaggio $\mathcal{L}$: le variabili che vi compaiono avranno valori assegnati prima di far girare $\mathcal{P}$ e avranno degli altri valori dopo che $\mathcal{P}$ si sarà fermato.

Possiamo quindi pensare a $\mathcal{P}$ come ad un marchingegno che effettua tale trasformazione nelle variabili.

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

$$\begin{array}{lcl} Z_1 & \leftarrow & f_1(Z_1, \ldots, Z_n) \\ \ldots & \ldots & \ldots \\ Z_n & \leftarrow & f_n(Z_1, \ldots, Z_n) \end{array}$$

Immaginiamo adesso di considerare il programma $\mathcal{P}$ come il blocco interno da ciclare di un'istruzione LOOP-END:

$$\begin{array}{l} \mathrm{LOOP\ V} \\ \quad \boxed{\mathrm{P}} \\ \mathrm{END} \end{array}$$

Dove V, in base alla seconda assunzione fatta non compare in $\mathcal{P}$: Sia $\mathcal{Q}$ tale nuovo programma. $\mathcal{Q}$ indurrà, a sua volta, la seguente trasformazione sulle $n+1$ variabili $Z_1, Z_2, \ldots, Z_n, V$:

$$\begin{array}{lcl} Z_1 & \leftarrow & g_1(Z_1, \ldots, Z_n, V) \\ \ldots & \ldots & \ldots \\ Z_n & \leftarrow & g_n(Z_1, \ldots, Z_n, V) \end{array}$$

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

Dimostrazione. Domandiamoci come facciamo a calcolare i valori della funzione g_i su $(z_1, \ldots, z_n, t+1)$. Andando a calcolare la corrispondente funzione f_i sui valori $g_1(z_1, \ldots, z_n, t)$, $\ldots, g_n(z_1, \ldots, z_n, t)$; cioè mediante il seguente meccanismo di ricorsione simultanea:

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

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:

$$(**) \quad \tilde{g}(z_1, \ldots, z_n, u) \;=\; [g_1(z_1, \ldots, z_n, u), \ldots, g_n(z_1, \ldots, z_n, u)]$$

per cui si ha che: $\tilde{g}(z_1, \ldots, z_n, 0) = [z_1, \ldots, z_n]$ e $\tilde{g}(z_1, \ldots, z_n, t+1) = [k_1, \ldots, k_n]$ dove i k sono ottenuti mediante la (*) e (**). Cioè si ha:

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

In cosa differisce l'ultima scrittura che riguarda le 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 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 3 *Sia $\mathcal{P}$ un programma LOOP che contiene solo le variabili $z_1, \ldots, z_n$. $\mathcal{P}$ trasformi i valori delle variabili $z_1, \ldots, z_n$ secondo lo schema*

$$\begin{array}{lcl} Z_1 & \leftarrow & f_1(Z_1, \ldots, Z_n) \\ \ldots & \ldots & \ldots \\ Z_n & \leftarrow & f_n(Z_1, \ldots, Z_n) \end{array}$$

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

Dimostrazione. Procediamo per induzione. Ammettiamo in primo luogo che sia $\mathcal{P} \in \mathcal{L}_0$. Allora $\mathcal{P}$ non può contenere istruzioni ciclo e le uniche istruzioni che può utilizzare sono: V $\leftarrow$ 0, V $\leftarrow$ V′, V $\leftarrow$ V+1. Perciò, il programma a ogni passo non puo fare altro che azzerare il valore di una variabile, porre una variabile uguale al valore di un altra variabile oppure incrementare di 1 il valore di una variabile (ovviamente ognuna di queste operazioni puo essere effettuata solo un numero finito di volte). Quindi le funzioni calcolate da questo tipo di programmi possono assumere soltanto una di queste 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 $\mathcal{L}_n$, vogliamo mostrare che lo è anche per un arbitrario programma $\mathcal{P}$ di $\mathcal{L}_{n+1}$.

Un programma di $\mathcal{L}_{n+1}$ si può decomporre in una serie di blocchi successivi ciascuno dei quali forma un programma appartenente ad $\mathcal{L}_n$, eventualmente inseriti in un ciclo LOOP-END. Indichiamo con la lettera $\mathcal{P}_i$ questi ultimi e con le lettere $\mathcal{Q}_i$ 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 $\mathcal{P}_i$ 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 una 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
...

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

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

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

Che è una trasformazione del tipo di quelle che abbiamo discusso, $\mathcal{Q}$ contiene solo variabili locali $Z_1, \ldots, Z_m$ con $k \leq s \leq m$.

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.

Proposizione 4 *Le funzioni calcolabili da programmi di $\mathcal{L}$ sono ricorsive primitive.*

Dimostrazione. Unendo i risultati delle due proposizioni precedenti.

3 La profondità di nidificazione come misura di complessità

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

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

Dunque, se $\mathcal{P}$ 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 $\mathcal{P}$ definito nel modo precedentemente detto.

Fatto: *Esiste un programma che calcola T_p e che ha una profondità di nidificazione non maggiore di $\mathcal{P}$.* In simboli: Se $\mathcal{P} \in \mathcal{L}_n$ allora $T_{\mathcal{P}} \in \mathcal{L}_n$

Verifica. Basta modificare il progamma $\mathcal{P}$ inserendo un contatore T che aumenta di una unità ogni volta che viene eseguita una delle istruzioni $V \leftarrow 0, V \leftarrow V', V \leftarrow V + 1$.

Questo può realizzarsi molto semplicemente ponendo un ìstruzione $T \leftarrow T + 1$ subito dopo ciascuna delle istruzioni dei tipi elencati sopra che sono presenti nel programma $\mathcal{P}$. È evidente che questo nuovo programma ha la stessa profondità di nidificazione di $\mathcal{P}$.

3.1 Una famiglia di funzioni e la loro velocità di crescita

Vogliamo adesso limitare superiormente i tempi di calcolo di varie funzioni. Ricordiamo la notazione $g^{(n)}(x) = \underbrace{g(g(\ldots(g(x))}_{m \text{ volte}})$ (composizione di g con se stessa n volte)

e poniamo $g^{(0)}(x) = x$.

Definiamo adesso la seguente famiglia di funzioni:

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

È facile verificare che

$$\begin{aligned} f_1(x) &= 2x \text{ (con } x \neq 0) \\ f_2(x) &= 2^x \\ f_3(x) &= \underbrace{2^{2^{\cdots^2}}}_{\text{x volte}} \end{aligned}$$

Vogliamo adesso studiare alcune proprietà di questa famiglia di funzioni.

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

Dimostrazione. Ricordiamo che $f_{n+1}(x) = f_n^{(x)}(1)$, allora $f_{n+1}(x+1) = f_n^{(x+1)}(1) = f_n(f_n^{(x)}(1)) = f_n(f_{n+1}^{(x)})$.

Lemma 2. $f_0^{(k)}(x) \geq k$ dove $f_0(x) = \begin{cases} x+1 & \text{Se } x = 0 \text{ oppure } x = 1 \\ x+2 & \text{altrimenti} \end{cases}$.

Dimostrazione. Per induzione su k

- Per $k = 0$ is ha $f_0^{(0)}(x) = x \geq 0$
- Assumiamo adesso il risultato valido per k vogliamo dimostrare che è valido per $k+1$.
  1. $f_0^{(k+1)}(x) = f_0(f_0^{(k)}(x))$
     poiché $f_0(x) \geq x+1$ per ogni x (per definizione di f_0) si ha che
  2. $f_0(f_0^{(k)}(x)) \geq f_0^{(k)}(x) + 1$ 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 le (1), (2) e (3): $f_0^{(k)}(x) \geq k$

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 che il risultato sia vero per $n = k$, cioè che $f_k(x) > x$, *per ogni* x, e 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 $x = m$

$$\begin{array}{rcll} f_{k+1}(m) & = & f_k(f_{k+1}(m)) & \text{(per il lemma 1)} \\ & & f_{k+1}(m) & \text{(per l'ipotesi di induzione su } k\text{)} \\ & & > m & \text{(per l'ipotesi di induzione su } x\text{)} \end{array}$$

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

Lemma 4. $f_n(x+1) > f_n(x)$

Dimostrazione.

- per $n = 0$ discende immediatamente dalla definizione

$$f_0(x+1) > f_0(x) \quad \left(\begin{array}{c} (x+1)+1 \\ (x+2)+1 \end{array} \right) > \left(\begin{array}{c} x+1 \\ x+2 \end{array} \right)$$

- per induzione su n abbiamo che:

$$f_n(x+1) \underset{\substack{\uparrow \\ \text{lemma 1}}}{=} f_{n-1}(f_n(x)) \underset{\substack{\uparrow \\ \text{lemma 3}}}{>} f_n(x)$$

Lemma 5. $f_{n+1}(x) \geq f_n(x)$

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 $f_{n+1}(x+1) > f_n(x)$.

Si ha che

$$f_{n+1}(x+1) \underset{\substack{\uparrow \\ \text{Lemma 1}}}{=} f_n(f_{n+1}(x)) \underset{\substack{\text{lemma 4 (tenendo presente} \\ \text{che } f_{n+1}(x) \geq x+1 \\ \text{per il lemma 3)}}}{\geq} f_n(x+1).$$

Lemma 6. $f_n^{(k+1)}(x) > f_n^{(k)}(x)$

Dimostrazione. $f_n^{(k+1)}(x) = f_n(f_n^{(k)}(x)) \underset{\substack{\uparrow \\ \text{Lemma 3}}}{>} f_n^{(k)}(x)$

Osservazione. Finora abbiamo verificato, quindi, che $f_n^{(k)}(x)$ è crescente sia al crescere della variabile x sia al crescere dell'indice n sia al crescere del numero di volte k in cui viene iterata la sua applicazione a se stessa. Anzi, su x e k è strettamente crescente. Verifichiamo adesso altre proprietà di crescita.

Lemma 7. $f_n^{(k+1)}(x) \geq 2f_n^{(k)}(x)$ (per $n \geq 1$)

Dimostrazione.

- per $k = 0$

$$f_n^{(1)}(x) = f_n(x) \underset{\text{Lemma 5}}{\uparrow} > f_1(x) \underset{\text{Definizione di } f_n}{\uparrow} = 2x \underset{\text{Definizione di } f_0}{\uparrow} = f_n^{(k)}(x)$$

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

$$f_n^{(k+2)}(x) = f_n^{(k+1)}(f_n(x)) \underset{\text{ipotesi di induzione}}{\uparrow} \geq 2 \cdot f_n^{(k)}(f_n(x)) = 2f_n^{(k+1)}(x)$$

Lemma 8. $f_n^{(k+1)}(x) \geq f_n^{(k)}(x) + x$ (per $n \geq 1$)

Dimostrazione.

- per $k = 0$ si ha che:
  $f_n^{(1)}(x) \geq 2f_n^{(0)}(x) = f_n^{(0)}(x) + f_n^{(0)}(x) = f_n^{(0)}(x) + x$
- per $k > 0$ si ha:

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

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

$$\begin{aligned} f_1^{(1)}(x) &= f_1(f_1^{(k)}(x) \underset{\substack{\text{Lemma 4 e ipotesi} \\ \text{induzione}}}{\uparrow} \geq f_1(2^k \cdot x) \underset{\text{lemma 7}}{\uparrow} \geq 2 \cdot f_1^{(0)} \cdot (2^k \cdot x) \\ &= 2 \cdot 2^k \cdot x = 2^{k+1} \cdot x \end{aligned}$$

Schema riassuntivo delle proprieta di crescita della famiglia di funzioni definita da

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

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

Lemma 8. $f_n^{(k+1)}(x) \geq f_n^{(k)}(x) + x$

Lemma 9. $f_1^{(k)}(x) \geq 2^k \cdot x$

3.2 Proprietà di limitazione alla crescita

Desideriamo adesso studiare alcune proprieta dei programmi di $\mathcal{L}$ relativamente alla crescita dei tempi di calcolo e dei valori assunti dalle variabili al crescere della profondità di nidificazione. Ricordiamo che all'inizio di questo paragrafo abbiamo definito il tempo di calcolo di un programma $\mathcal{P} \in \mathrm{L}_n$ come il numero totale di istruzioni di assegnazione e incremento che vengono eseguite; abbiamo, poi, trovato un semplice collegamento tra la struttura del programma e il tempo di calcolo. In base alle proprieta della famiglia di funzioni f_n che abbiamo appena stabilito, possiamo stabilire un altro legame tra il tempo di calcolo di un programma e la crescita dei valori delle variabili presenti nel programma stesso. Una proprietà che ci sarà utile tra poco.

Sia u il valore massimo in ingresso al programma $\mathcal{P}$ e supponiamo che si sappia che $T_p(x_1, \ldots, x_m) \leq f_n^{(x)}(u)$. 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 far aumentare di molto il valore della singola variabile ma non aumenta il valore massimo delle variabili)
- Essere incrementati di una unità

Il caso più sfavorevole riguardo all'aumento del massimo valore delle variabili è proprio dato da un aumento di una unità, ripetuto, della variabile che ha il valore massimo. L'istruzione di assegnazione, infatti, come già osservato, pur potendo modificare di molto, in un colpo solo, il valore di una singola variabile, non modifica in alcun modo il valore massimo delle variabili esaminate. Quindi se è $T_p(x_1, \ldots, x_m) \leq f_n^{(x)}(u)$, 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' sara:

$$u' \leq u + T_p(x_1, \ldots, x_m) \leq u + f_n^{(x)}(u) \underset{\text{lemma 8}}{\leq} f_n^{(x)}(u)$$

Passiamo adesso a dimostrare la proposizione seguente che assieme a un suo corollario riportato piu avanti e solitamente indicato come **Teorema della limitazione alla crescita** (la crescita riferendosi sia al tempo di calcolo sia al valore che possono assumere le variabili e quindi di conseguenza, anche il valore assunto dalla funzione calcolata dal programma preso in considerazione)

Proposizione 5 *Sia $\mathcal{P} \in L_n$ 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 $\mathcal{P}$ non ha cicli e quindi (numero di istruzioni del programma) ma $T_p(x_1, \ldots, x_m) = k$ (per il lemma2)

- assumiamo adesso che il risultato sia vero per $n-1$ e sia $\mathcal{P}$ un generico programma di $\mathcal{L}_n$. Procediamo per gradi.

  - P appartenente ad $\mathcal{L}_n$, appartenga anche ad $\mathcal{L}_{n-1}$ allora avremo che:
    $T_p(x_1,\ldots,x_m) \underset{\text{ipotesi induttiva}}{\leq} f_{n-1}^{(k)}(u) \underset{\text{lemma 5}}{\leq} f_n^{(k)}(u)$
  - Sia adesso P il programma seguente

    LOOP V
    | Q |
    END

    con $\mathcal{Q} \in \mathcal{L}_{n-1}$, 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 $\mathcal{Q} \in \mathcal{L}_0$. In questo caso sara: $T_q = q$, per cui sarà $T_p(x_1,\ldots,x_m) = qv$ (dove v è il valore di V in LOOP V) ma sicuramente si avrà

    $$qv \underset{\substack{\text{lunghezza di } Q \\ \text{"ciclata" } v \text{ volte}}}{\leq} qu \underset{\substack{\text{per un } k \text{ op-} \\ \text{portuno}}}{\leq} 2^k \cdot u \underset{\text{lemma 9}}{\leq} f_1^{(k)}(u)$$

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

    $$(*) \quad T_{\mathcal{Q}}(x_1,\ldots,x_m) \leq f_{n-1}^{(j)}(u) \quad \text{per qualche } j$$

    Un calcolo di $\mathcal{P}$ è ottenuto facendo girare $\mathcal{Q}$ per v volte.

Dobbiamo però tenere conto del fatto che ad ogni nuovo ciclo i valori delle variabili possono essere aumentati.

Dobbiamo perciò trovare valori che limitino la crescita possibile delle variabili dopo ogni esecuzione di Q nella (*) la li mitazione 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 *limitazione alla crescita* dei valori delle variabili che nel nostro caso diviene:

$$\boxed{T_Q(x_1,\ldots,x_m) \leq f_{n-1}^{(j)}(u) \Rightarrow u' \leq u + f_{n-1}^{(j)}(u) \leq f_{n-1}^{(j+1)}(u)}$$

dove u' è il nuovo valore massimo delle variabili.

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

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

Il tempo di calcolo totale è quindi limitato da:

$$\begin{array}{lcl} f_{n-1}^{(j)}(u) + f_{n-1}^{(j)}(f_{n-1}^{(j+1)}(u)) & = & f_{n-1}^{(j)}(u) + f_{n-1}^{(j+1)}(f_{n-1}^{(j)}(u)) \quad \underset{\substack{\text{considerando} \\ f_{n-1}^{(j+1)}(u) \text{ variabile}}}{\overset{\text{Lemma 8}}{\leq}} \\ f_{n-1}^{(j+2)}(f_{n-1}^{(j)}(u)) & = & f_{n-1}^{(2j+2)}(u) \end{array}$$

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 $\mathcal{Q}$ da: $f_{n-1}^{(j+1)}(u)$

dopo 2 giri di $\mathcal{Q}$ da: $f_{n-1}^{(2j+3)}(u)$

Dobbiamo quindi aggiungere all'esponente di f un numero pari a $j+2$ per volta. Un buon lmite (per eccesso) per entrambi T e le variabili dopo u esecuzioni di $\mathcal{Q}$ è dato da:

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

(l'istruzione LOOP ci dice di eseguire $\mathcal{Q}$ v volte, moi non conosciamo v ma sappiamo che minore o uguale a u) Allora abbiamo che:

$$\begin{array}{lllll} T_P(x_1,\ldots,x_m) & \leq & T_{Q[u]}(x_1,\ldots,x_m) & \leq & f_{n-1}^{(u\cdot(j+2))}(u) \\ & \underset{\text{Lemma 3 e 4}}{\leq} & f_{n-1}^{(u\cdot(j+2))}(f_n(u)) & \underset{\text{definizione di } f_u(u)}{=} & f_{n-1}^{(u\cdot(j+2))}(f_{n-1}^{(u)}(1)) \\ & = & f_{n-1}^{(u\cdot(j+2)+u)}(1) & = & f_{n-1}^{(u\cdot(j+3))}(1) \\ & \underset{\text{Lemma 6}}{\uparrow\leq} & f_{n-1}^{(u\cdot 2^{(j+2)})}(1) & \underset{\text{Lemma 9}}{\uparrow\leq} & f_{n-1}^{(f_1^{(j+2)}(u))}(1) \\ & \underset{\text{Lemma 5 e 6}}{\uparrow\leq} & f_{n-1}^{(f_n^{(j+2)}(u))}(1) & \underset{\text{Definizione di } f_n}{\uparrow=} & f_n(f_n^{(j+2)}(u) \\ & = & f_n^{(j+3)}(u) & = & f_n^{(k)}(u) \end{array}$$

posto $k = j+3$

Abbiamo così dimostrato il teorema per i programmi $\mathcal{P}$ della forma:

LOOP V
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 a $\mathcal{L}_{n-1}$ oppure un sottoprogramma della forma considerata prima

LOOP V
Q
END

con $\mathcal{Q} \in \mathcal{L}_{n-1}$.

Le variabili di uscita di ciascun sottoprogramma saranno le variabili di ingresso del sottoprogramma successivo. Avremo quindi che:

$$\begin{aligned} T_P(x_1,\ldots,x_m) \quad \leq \quad & f_n^{(k_1)}(u) + f_n^{(k_2)}(f_n^{(k_1)}(u)) + \\ & f_n^{(k_3)}(f_n^{(k_2)}(f_n^{(k_1)}(u))) + \ldots \\ & + f_n^{(k_s)}(\ldots(f_n^{(k_1)}(u))) \leq f_n^k(u) \end{aligned}$$

per un opportuno k, applicando il lemma 8 dopo aver osservato che il primo termine è l'argomento del secondo e così via. Volendo calcolare esplicitamente il k osserviamo cosa succede nel caso di una somma di soli tre termini.

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

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

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

Allora abbiamo che:

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

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

Osserviamo però che, per il lemma 4,

$$f_n^{(k_3)}(f_n^{(k_2)}(f_n^{(k_2)}(u))) < f_n^{(k_3)}(f_n^{(k_2+1)}(f_n^{(k_2)}(u)))$$

poiché $f_n^{(k_2)}(f_n^{(k_2)}(u)) < f_n^{(k_2+1)}(f_n^{(k_2)}(u))$ per il lemma 6.

Allora avremo che

$$F < f_n^{(k_2+1)}(f_n^{(k_2)}(u)) + f_n^{(k_3)}(f_n^{(k_2)}(f_n^{(k_2)}(u)))$$

possiamo ora applicare il lemma 8 ed ottenere che

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

esaminiamo come compaiono gli "esponenti": sono tutti aumentati di 1 tranne il primo. Ci si convince facilmente che il meccanismo resta invariato per una somma di s termini per cui si ha che il nostro T_P è maggiorato da:

$$f_n^{[(k_s+1)+(k_{s-1}+1)+(k_{s-2}+1)+\ldots+(k_3+1)+(k_2+1)+k_1]}(u) = f_n^{(k_s+k_{s-1}+k_{s-2}+\ldots+k_3+k_2+k_1)}(u)$$

utilizzando ancora una volta la proprietà di limitazione alla uscita avremo allora il **Corollario** Se $g \in \mathcal{L}_n$ allora esiste una costante k tale che

$$g(x_1,\ldots,x_m) \leq f_m^{(k)}(\max(x_1,\ldots,x_m))$$

3.3 $\mathcal{L}_n$ è una gerarchia

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

è effettivamente cosí.

Proposizione 6 *Per $n \geq 1$, $f_n \in \mathcal{L}_n$*

Dimostrazione. Per induzione su n:

- Per $n = 1$ $f_1(0) = 1$

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

  f_1 è calcolata dal programma:

```
Y ← Y + 1                    ← (questa per avere in uscita 1 quando x = 0)
LOOP X
        X ← X+1
        Y← X
END
```

- Scriviamo adesso un programma che calcoli f_{k+1}. Per definizione quindi

```
Y← Y+1
Z←Z+1
LOOP X
        Y ← f_k(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 aggiunto un nuovo ciclo e l'ipotesi di induzione $f_k \in \mathcal{L}_k$ ci garantisce anche che possiamo scrivere macro $Y \leftarrow f_k(Z)$, perché f_k è ricorsiva primitiva.

Per ottenere il nostro risultato ci resta da mostrare che $f_k \notin \mathcal{L}_{n-1}$.

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 un *maggiorante* di g ($f \succ g$) se $f(x) > g(x)$ *quasi ovunque.*

Proposizione 7 *($f_{n+1} \succ f_n^k$)* per ogni n e k.

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

$f_1(x) = 2x$

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

Sia adesso $n \neq 0$ e consideriamo induzione su k, per $k = 0$ si ha che:

$$f_{n+1}(x) \underset{\substack{\uparrow \\ \text{Lemma 3}}}{>} x \underset{\substack{\uparrow \\ \text{definizione di } f^k}}{=} f_n^{(0)}(x)$$

Possiamo quindi supporre $f_{n+1} \succ f_n^k$ e vogliamo mostrare che è anche $f_{n+1} \succ f_n^{k+1}$.

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

$$\begin{array}{ll} f_n^{(k+1)}(x) & \underset{\substack{\uparrow \\ \text{Lemma 4 (da un certo punto in poi)}}}{<} \\ f_n^{(k+1)}(2n-4) & \underset{\substack{\uparrow \\ \text{per definizione di } f_1}}{=} \\ f_n^{(k+1)}(f_1(x-2)) & \underset{\substack{\uparrow \\ \text{Lemma 4 e 5}}}{<} \\ f_n^{(2)}(f_n^{(k)}(x-2)) & \underset{\substack{\uparrow \\ \text{ipotesi di induzione da un certo punto in poi)}}}{\leq} \end{array}$$

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

$$f_n^{(2)}(f_n^{n-2)}(1)) =$$

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

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

cioè:

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

Possiamo adesso dimostrare la

Proposizione 8 *$f_{n+1} \in \mathcal{L}_{n+1}$ ma $f_{n+1} \notin \mathcal{L}_n$*

Dimostrazione

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

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

ma per la proposizione appena mostrata si ha che;

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

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

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

Osservazione. Sottolineiamo alcuni dei risultati trovati:

- *tutte* le f_n sono ricorsive primitive.
- $\mathcal{L}_n$ è *una gerarchia* e quindi aumentando la profondità di nidificazione riusciamo a calcolare effettivamente più cose. Questo dà una risposta alla domanda iniziale se ci sia effettivamente bisogno di potere aumentare indefinitamente la profondita di nidificazione per calcolare tutte le funzioni ciclo-calcolabili. Per riprendere l'esempio portato all'inizio, con programmi di profondità di nidificazione 137.731 possiamo realmente fare più cose di quelle che si possono fare con programmi che hanno al massimo profondità di nidificazione 137.730.
- Dalla proposizione 8, discende, infine, la risposta (negativa) alla domanda se sia possibile scrivere un programma per la moltiplicazione che abbia profondita di nidicazione 1.

4 La funzione di Ackermann

Mostreremo adesso l'esistenza di una funzione calcolabile ma non ricorsiva primtiva. 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 intendiamo nel senso di calcolabile in $\mathcal{S}$).

4.1 La funzione di Ackermann non è ricorsiva primitiva

La funzione che andremo adesso ad esaminare è la funzione di Ackermann di due variabili definita da: $A(i,x) = f_i(x)$ e che gode delle proprietà:

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

Proposizione 9 *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 primtiva. Allora, poiché sappiamo già che le funzioni ricorsive primitive sono calcolabili da un programma-ciclo, esisterà un m tale che $g \in \mathcal{L}_n$ ma, in base ai teoremi sulla limitazione alla crescita, sappiamo ancora che:

$$f_m^{(k)}(x) < f_{m+1}(x) \qquad \text{quasi ovunque } g \in \mathcal{L}_m \Rightarrow \exists k \text{ tale che } g(x) \leq f_m^{(k)}(x)$$

per cui sarà

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

Sia n_0 un intero $> m+1$ per cui la diseguaglianza 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 $n_0 > m+1$ si avrà $f_{n_0}(n_0) > f_{m+1}(n_0)$, che è una contraddizione.

Abbiamo così dimostrato che $A(x,x)$ *non* è ricorsiva primitiva.

Cosa 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.

4.2 La funzione di Ackermann è calcolabile in $\mathcal{S}$

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

Usiamo una particolare memoria detta pila in cui immagazzinare gli argomenti di sinistra mentre stiamo valutando quelli di destra.
La pila sarà indicata da $(a_1, \ldots, a_m)$ e si converrà che a_1 è l'elemento più in alto nella pila e, in generale, ai sta più in alto di a_{i+1}.

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

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 espressione $A(i,0)$ e $A(0,x)$ si sanno calcolare immediatamente, per cui, successivamente:

3. si prende il valore più in alto della pila e si opera nuovamente come in 1 sulla nuova espressione.

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 sulla lunghezza della pila stessa e la seconda contiene i valori che abbiamo conservato nella pila.

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

$L \leftarrow L+1$ (aumenta di 1 la lunghezza)
$S \leftarrow < q, s >$ (codifica mediante la $<,>$il nuovo q)

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

$L \leftarrow L - 1$ (diminuisce di 1 la lunghezza)
$i \leftarrow l(S)$ (prende l'elemento sinistro di $S =<,>$ che è quello voluto)
$S \leftarrow r(S)$ (rimetti in S ciò che resta, cioè il lato destro di $<,>$)

Presentiamo adesso un programma di $\mathcal{S}$ che calcola la funzione di Ackermann $A(i,x)$.

Ricordiamo la definizione di $A(i,x)$

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

Questo è il programma:

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

Sono possibili quattro casi:

1. $A(i,0) = 1$ realizzato dalle istruzioni $1 \rightarrow 2 \rightarrow 3 \rightarrow 18$
2. $A(0,1) = 2$ realizzato dalle istruzioni $1 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 3 \rightarrow 18$
3. $A(0,x) = 2x$ per $x > 1 : A \rightarrow D \rightarrow C$ (se non c'è niente nella pila) $1 \rightarrow 10 \rightarrow 11 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 18$

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 della pila

Possiamo visualizzare il funzionamento del programma anche in questo modo:

- Il gruppo di istruzioni 1-2 calcola $A(i,0) = 1$ oppure rinvia al gruppo [D] e successive (che potremmo chiamare *gruppo di 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 $x > 1$)
  - $(x = 1, i = 1) \to$ (calcolo si $A(0,1)$)
- il gruppo [E] 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 $(i \neq 0, x \neq 0)$.

References

[1] A. Meyer and D. M. Ritchie (1967), The complexity of loop programs, *Proceedings 22nd A. C. M. National Meeting*, pages 465-469, Washington (USA).

[2] R. Péter (1967), *Recursive functions*, Academic Press.

[3] M. Davis and E. Weyuker (1983), *Computability, complexity and languages*, Academic Press.