*DISPENSE DEL CORSO DI*

*INFORMATICA TEORICA – Parte 1*

# INTRODUZIONE ALLA CALCOLABILITA'

Alberto Bertoni

## INDICE

# 1. RICHIAMI DI NOZIONI SUGLI INSIEMI - NOTAZIONI

Indichiamo con maiuscole (A, B, ...) le variabili sugli insiemi, con minuscole (x, y, a, ...)le variabili sugli elementi, salvo indicazioni contrarie. Il simbolo della relazione di appartenenza è $\in$ ("x $\in$ A" sarà letto "x appartiene ad A").
Dati due insiemi A, B, una **funzione** *f* di dominio A e codominio B è una legge che ad ogni elemento a $\in$ A associa un elemento b $\in$ B. Scriveremo *f* : A -> B, evidenziando il nome di funzione *f*, il dominio A e il codominio B; scriveremo

$$b = f(a) \quad o \quad a \overset{f}{\longmapsto} b$$

Una funzione *f* si dice **iniettiva** sse $x_1 \neq x_2$ implica $f(x_1) \neq f(x_2)$. Immagine di *f* è $Im_f \equiv \{y \mid \exists\ x \in A\ (\ y = f(x))\ \}$. Una funzione si dice **suriettiva** sse la sua immagine coincide con il codominio.
Una **corrispondenza biunivoca** (o funzione **biiettiva**) è una funzione *f* : A -> B tale che $\forall\ y \in B\ \exists!\ x \in A\ (\ y = f(x))$ ("$\exists!$" significa "Esiste un unico").Una funzione è biiettiva se e solo se è sia iniettiva che suriettiva.
Data una funzione biiettiva *f* : A -> B, si dice **inversa** di *f* la funzione *f*^(-1) : B -> A definita da x = *f*^(-1)(y) sse y = *f*(x).
Date due funzioni *f* : A -> B, *g* : B -> C (il codominio della prima coincide con il dominio della seconda) diciamo **composizione**

$$g \text{ o } f\text{: A -> C}$$

la funzione definita tramite la legge (*g* o *f*)(x) = *g*(*f*(x)). La composizione di tre funzioni, quando è definita, gode della proprietà associativa: (*f* o *g*) o *h* = *f* o (*g* o *h*).
La funzione $I_A$ : A -> A definita da $\forall$ a ($I_A$(a) = a) si chiama funzione **identità** su A.
Se *f* : A -> B, allora $I_B$ o *f* = *f*, *f* o $I_A$ = *f*.
Se inoltre *f* : A -> B è una corrispondenza biunivoca, allora *f* o *f*^(-1)= $I_B$, *f*^(-1) o *f* = $I_A$.

Esempio: Sia *N* = (0, 1, 2, ..., n, ... } l'insieme dei naturali (non negativi). Le funzioni successore (**succ**) e predecessore (**pred**) sono definite:

succ : *N* -> *N*, succ(x) = x + 1

.

pred : $N$ -> $N$, pred(x) = x - 1 se x = 0 allora 0
altrimenti x - 1

succ è iniettiva (non suriettiva); pred è suriettiva (non iniettiva).

Una nozione che verrà spesso usata in seguito è quella di **funzione parziale**. Una funzione parziale $f$ : A -> B è una legge che a certi elementi di A associa un elemento di B. Se all'elemento a $\in$ A viene associato un elemento di B, diremo che $f$ è **definita** su a, altrimenti verrà detta **indefinita**. **Dominio** (di definizione) di f è

$$\{x \mid x \in A,\ f(x) \text{ definita}\}$$

Una funzione parziale $f$ : A -> B può essere pensata come una funzione ("totale", in contrapposizione a "parziale")
~
$f$: A -> B U {⊥}, dove ⊥ è un simbolo (distinto) di indefinito:
~
$f(x)$ = se f indefinita su x allora ⊥
altrimenti $f(x)$

Analizziamo ora alcune generali costruzioni insiemistiche:

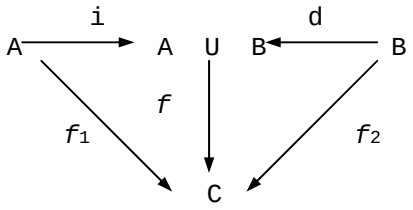
**COPRODOTTO** (o unione disgiunta): Dati due insiemi A, B, diciamo **unione disgiunta** di A e B l'insieme A U B unione di due coppie isomorfe e disgiunte di A e B, cioè

$$A \cup B = \{(a, 1) \mid a \in A\} \cup \{(b, 2) \mid b \in B\}.$$

Le due iniezioni i : A -> A U B, j : B -> A U B sono definite da i(a) = (a, 1), j(b) = (b, 2). Date due funzioni

$f_1$ $f_2$
A ---> C <--- B

C'è un'unica funzione $f$ : A U B -> C tale che il diagramma seguente




commuta (cioè: $f$ o $i = f_1$, $f$ o $j = f_2$).

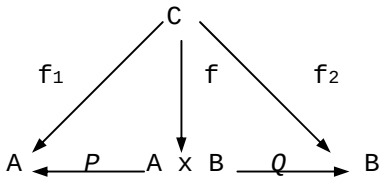
**PRODOTTO**: Dati due insiemi A, B, il loro **prodotto** A x B è l'insieme di coppie ordinate di elementi di A e di B cioè:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

Le due proiezioni $P$ : A x B -> A; $Q$: A x B -> B sono definite da: $P(x, y) = x$, $Q(x, y) = y$. Date due funzioni

$$A \xleftarrow{f_1} C \xrightarrow{f_2} B$$

c'è un'unica funzione $f$ : C -> A x B tale che il diagramma seguente commuta:
1



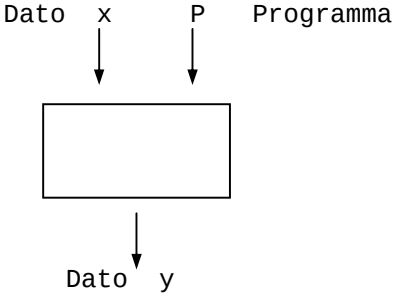

(cioè $P$ o $f = f_1$, $Q$ o $f = f_2$)

**FUNZIONI**: Dati due insiemi A, B, diciamo **Funzioni** di A in B (o esponenziale B^A) l'insieme B^A (o B -> A) tale che:

$$B^A = \{f \mid f : A \to B\}$$

La funzione valutazione $w$ : B^A x A -> B è definita come segue:
$w(f, a) = f(a)$.

Esempio: analizziamo informalmente il comportamento di un sistema di calcolo:




sul dato di ingresso x, il programma P dà luogo ad una sequenza di passi, che può terminare e dar luogo al dato y di uscita, oppure può non terminare, nel qual caso diciamo che il valore della computazione è $\perp$.

Quindi il comportamento è opportunamente descritto da una funzione $C$: **Dati** x **Programmi** -> **Dati** U $\{\perp\}$, dove **Dati** è l'insieme dei dati, **Programmi** è l'insieme dei programmi.

**CARDINALITA'**: Dati due insiemi A, B, diciamo che essi sono **isomorfi**, scrivendo $A \cong B$, se c'è una corrispondenza biunivoca

$$f: A \to B$$

E' facile mostrare che ≅ è una relazione di equivalenza. Chiameremo la classe quoziente classe della **cardinalità**. In particolare, *cardinalità* del numerabile è la classe di equivalenza contenente

$$N = \{0, 1, 2, ..., n, ...\}$$

**TEOREMA**: $N \sim\neq N^N$

***Dimostrazione***: Se, per assurdo, $N^N$ fosse numerabile, potremmo porre $N^N \equiv \{f_k \mid k \geq 0\}$.
$f_k$ è la funzione $f_k : N \to N$ associata da una data corrispondenza biunivoca all'intero k.
Consideriamo ora la funzione $\varphi(n) = f_n(n) + 1$. Abbiamo:
$\varphi : N \to N$, quindi $\varphi \in N^N$. Esiste allora $\overline{k}$ per cui: $\varphi = f_{\overline{k}}$
Allora $\varphi(\overline{k}) = f_{\overline{k}}(\overline{k}) + 1 = f_{\overline{k}}(\overline{k})$, assurdo. ♦

Poiché, in un sistema digitale, sia i dati che i programmi devono essere specificati con una quantità finita di informazione (usando un numero finito di bits), ogni dato (e ogni programma) può essere descritto con un intero. Cioè:

**Dati** ≅ N; **Programmi** ≅ N.

L'insieme delle funzioni computabili da un dato sistema, descritto da

$$C : \textbf{Dati} \times \textbf{Programmi} \to \textbf{Dati} \cup \{\perp\}$$

è allora:

**Funzioni computabili** ≅ $\{f \mid f : N \to N \cup \{\perp\},$
$\exists P \in$ **Programmi** $(f(x) = C(x, P))\}$

Allora:
**Funzioni computabili** ≅ **Programmi** ≅ $N \sim\neq (N \cup \{\perp\})^N$
Esistono quindi funzioni non computabili, e il problema di caratterizzare la classe di quelle computabili è ben posto.

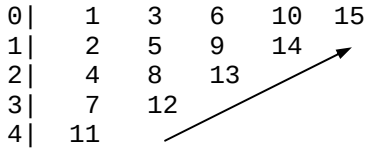
## 2. FUNZIONE COPPIA

Abbiamo in precedenza mostrato che $N^N \sim\neq N$. Mostriamo ora che $N \times N \cong$ N, cioè che coppie ordinate di interi si possono porre in corrispondenza biunivoca con gli interi.

**TEOREMA**: Esiste una corrispondenza biunivoca
$< , > : N \times N \to N+$

**Dimostrazione**: Basta considerare la seguente tabella:

| x \ y | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|




E' chiaro come, estendendo tale legge, a ogni coppia (x, y) corrisponda un preciso intero positivo n = < x, y > e viceversa. Non è difficile ottenere esplicitamente la funzione < x , y > come composizione di operazioni aritmetiche; infatti:

a) Lungo l'asse x, è

$$<z, 0> = 1 + 1 + 2 + \ldots + z = 1 + \frac{z(z+1)}{2}$$

b) Lungo la diagonale, è

$$<z - y, y> = <z, 0> + y = 1 + \frac{z(z+1)}{2} + y$$

Posto x = z – y, otteniamo:

$$<x, y> = 1 + \frac{(x+y)(x+y+1)}{2} + y$$

♦

Non è difficile costruire i programmi per le due proiezioni

sin: $N+ \to N$ e des: $N+ \to N$

caratterizzate dal fatto che:

sin(<x, y>) = x e des(<x, y>) = y

E' pure immediato rappresentare coppie di interi con interi: basta porre [x, y] = <x, y> -1; [,]; $N \times N \to N$ è chiaramente una corrispondenza biunivoca.
Con tecniche analoghe, è possibile mostrare che le usuali strutture informative possono essere rappresentate con interi, per cui, al nostro livello di generalità, è ragionevole porre **Dati** = $N$

Esempio 1: Vettori di interi n-dimensionali

Essendo fissata la dimensione n ($n \geq 2$), vogliamo descrivere n-uple di interi con interi. Una semplice soluzione è quella di porre:
$[x_1, x_2, \ldots, x_n] = [x_1, [x_2, \ldots, [x_{n-1}, x_n]\ldots]]$,

usando una ripetuta composizione della funzione [,] sopra definita.
Le funzioni $Pro(k, [x_1, \ldots, x_k, \ldots, x_n] = x_k$ sono facilmente programmabili.

Esempio 2: Sequenze finite di interi

Descriveremo le sequenze con la notazione $<x_1, x_2, ..., x_m>$. Mentre nel caso dei vettori il numero di elementi è fisso (nel caso dei vettori n-dimensionali è n), le componenti di una sequenza sono in numero variabile. Una semplice codifica di interi con interi positivi è:

$$<x_1, x_2, ..., x_m> = < x_1 <x_2 <... <x_m, 0 > ... > >$$

usando una ripetuta composizione della funzione coppia < , >.
Anche in questo caso le funzioni

$$Pro(k, <x_1,...,x_m>) = \text{se } k > m \text{ allora } \perp \text{ altrimenti } x_k$$

sono facilmente programmabili.
Analogamente per la funzione $lung(<x_1,..., x_m>)=m$.
Un semplice programma per tale funzione è, per esempio:

```
Ingresso x
      L := 1
      while x ≠ 0 do {x := des(x)
                      L := L+1 }
Uscita  L
```

# 3. LINGUAGGI DI PROGRAMMAZIONE

Per la comprensione di un linguaggio di programmazione (si pensi, per fissare le idee, al PASCAL studiato nel corso TAMC) occorre distinguere almeno due aspetti: **sintassi** e **semantica**.
Da un punto di vista sintattico, un linguaggio di programmazione è un insieme (generalmente infinito) di programmi, dove un "programma" è una particolare sequenza finita di simboli di un dato alfabeto.
Da un punto di vista semantico, il programma è invece visto come insieme di regole che trasformano il dato di ingresso nell'eventuale dato di uscita. Si può, riassumendo, dire che il significato di un programma è la funzione parziale realizzata.

Non è obiettivo di questo corso una analisi adeguata dei concetti sopra esposti. Gli aspetti sintattici saranno prevalentemente studiati in " Linguaggi Formali e Compilatori " , quelli semantici (e/o pragmatici) in "Linguaggi Speciali di Programmazione" - e altri; è tuttavia necessaria una comprensione di questi elementi per procedere alla presentazione dei semplici linguaggi di programmazione (Linguaggio RAM (ridotto), Linguaggio while) che ci serviranno come introduzione della teoria della ricorsività.

# 4. LINGUAGGIO RAM (ridotto)

Un meccanismo di calcolo che presenta qualche affinità con quelli reali è quello delle macchine a registri (di cui RAM è l'acronimo della traduzione inglese Random Access Machines = Macchine ad Accesso Casuale). Ne diamo una versione semplificatissima (senza indirizzamento indiretto, istruzioni aritmetiche semplicissime, un unico tipo di salto condizionato). Per ulteriori dettagli si veda "Complessità di calcolo delle funzioni", Ausiello.

**SINTASSI**: Un programma RAM (ridotto) è una sequenza finita di istruzioni appartenenti a uno dei seguenti tre tipi:

(1°) $R_k$ <-- $R_k$ + 1

(2°) $R_k$ <-- $R_k$ – 1

(3°) IF $R_k$ = 0 THEN GOTO n

dove k, n sono interi non negativi.

Della sintassi precedente (informale) si può dare una formalizzazione nei termini della seguente grammatica (si conviene di separare con gli elementi di una sequenza):

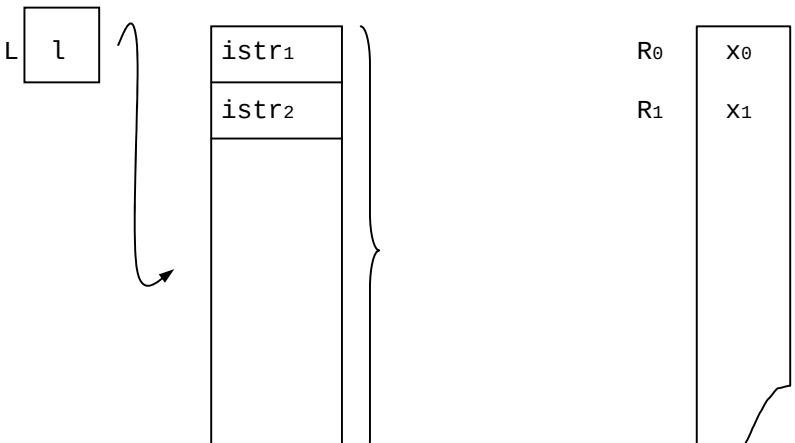
**GRAMMATICA**:

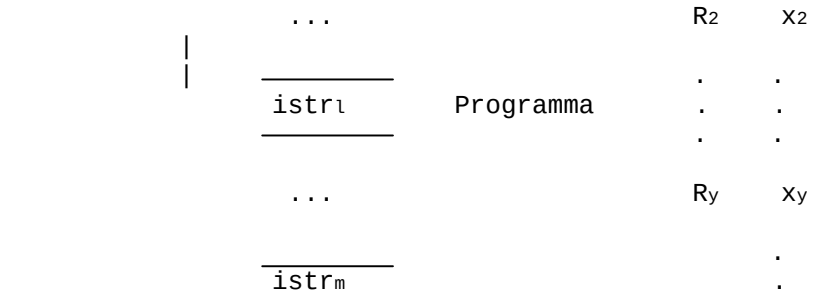
< istr > -> $R_k$ <- $R_k$ + 1 / $R_k$ - 1 /
IF $R_k$ = 0 THEN GOTO n ($k, n \geq 0$)
< prog > -> < istr > / < istr > ; < prog >

Conveniamo di scrivere le sequenze o in riga (separando con il simbolo ";" le istruzioni) o, equivalentemente, a capo.

Esempio: $R_0$ <- $R_0$ + 1; $R_0$ <- $R_0$ - 1 o, indifferentemente
$R_0$ <- $R_0$ + 1
$R_0$ <- $R_0$ – 1

**SEMANTICA OPERAZIONALE**:
daremo la semantica in termini delle operazioni di macchina che il compilatore fa corrispondere alle istruzioni e che, in fase di esecuzione, danno luogo a trasformazioni dei dati (da qui il termine "operazionale").






Informalmente: la macchina possiede infiniti registri

$$L, R_0, R_1, \ldots, R_j, \ldots$$

ognuno capace di contenere un intero, con L "contatore istruzioni".
I dati su cui si opera sono gli interi.
La macchina viene inizializzata ponendo 1 in L e il dato di ingresso x in $R_1$. Tutti gli altri registri sono azzerati.
L'esecuzione viene effettuata dalla Unità Centrale, che accede all'informazione contenuta nei registri con "accesso casuale". Le istruzioni vengono eseguite in sequenza (incrementando L di 1) a meno che l'istruzione di salto condizionato non alteri la sequenzialità.
Il programma termina quando il contenuto di L è 0. Il dato di uscita y è il contenuto di $R_0$. Più formalmente, la semantica può essere definita attraverso le seguenti nozioni:

*Def.1*: **Stato** è una funzione S : {L, $R_0$, $R_1$, $R_2$, ...} -> *N*
L'intero S($R_j$) è detto contenuto di $R_j$ nello stato S.
Intuitivamente, lo stato è una "fotografia" dei contenuti dei registri in un certo istante. Indichiamo con **Stati** l'insieme degli stati.

*Def.2*: **Stato finale** è uno stato S in cui S(L)=0.

*Def.3*: **Dato** è un qualsiasi intero non negativo.

Indichiamo con **Dati** l'insieme dei dati {0, 1, 2, ..., n, ...}

*Def.4*: **Inizializzazione** è la funzione in: **Dati** -> **Stati** definita come segue:

$$x \mid\!\!\rightarrow S_{in}$$

dove:

$S_{in}(L)=1$, $S_{in}(R_j)$ = Se j=1 allora x
altrimenti 0

Indichiamo ora con **Programmi** l'insieme dei programmi, indicando con |*P*|, per un programma *P*, il numero di istruzioni di *P*.

*Def.5*: **Stato Prossimo** è la funzione (parziale)

$\delta$: **Stati** x **Programmi** -> **Stati**

dove S'= $\delta$ (S, *P*) è definito dalla seguente procedura:

Caso S(L) = 0 allora S' è indefinito
Caso S(L) > |*P*| allora S'(L)=0, S'($R_j$)=S($R_j$) (j>=0)
Caso 0 < S(L) $\leq$ |*P*| allora:
Seleziona l'istruzione S(L) in *P*, diciamo Istr

Se Istr è del tipo "$R_k$ <- $R_k$ + 1"
S'(L) = S(L) + 1
S'($R_j$) = Se j = k allora S($R_j$)+1
altrimenti S($R_j$)

.

Se Istr è del tipo "$R_k$ <- $R_k$ - 1"
S'(L) = S(L) + 1 .
S'($R_j$) = Se j = k allora S($R_j$) - 1
altrimenti S($R_j$)

Se Istr è del tipo "IF $R_k$ = 0" then goto m"
S'(L) = S($R_k$) = 0 allora m
altrimenti S(L)+1
S'($R_j$) = S($R_j$)

*Def. 6*: **Sequenza di computazione** del programma *P* sul dato x è una successione {$S_0$, $S_1$, ..., $S_n$} di stati tali che:

(a) $S_0$ = in(x)
(b) $S_{e+1}$ = $\delta$ ($S_e$, P) (0 $\leq$ e < n)

*Def. 7*: **Sequenza di computazione accettata** dal programma *P* sul dato x è una sequenza di computazione < $S_0$, ..., $S_n$> dove $S_n$ è uno stato finale ($S_n$(L) = 0).

Siamo ora in grado di associare ad ogni programma *P* una precisa funzione parziale $\varphi_P$ : **Dati** -> **Dati** come segue (introducendo il simbolo $\perp$ per "indefinito"):

*Def.8*: **Funzione computata** del programma *P* è la funzione

$\varphi_P$ : **Dati** -> **Dati** U {$\perp$}

definita da:

$$\varphi_P(x) = \begin{cases} y \text{ se c'è una sequenza di computazione} \\ \quad < S_0, \ldots, S_n > \text{ accettata da P} \\ \quad \text{sul dato x ed inoltre } y = S_n(R_0) \\ \perp \text{ altrimenti} \end{cases}$$

Con la nozione di funzione computata da *P* abbiamo finalmente dato la semantica (il significato) di *P*, attraverso la funzione $\varphi_P$.
Concludendo, la sintassi ci ha permesso di definire il linguaggio di programmazione Programmi, mentre la semantica ci permette di definire il sistema di programmazione $\{\varphi_P\}$ (da taluni egualmente chiamato linguaggio di programmazione).

*Osservazione 1*: $\varphi_P$ è in generale una funzione parziale, essendo possibili sequenze di computazioni infinite. Vedremo che tale fenomeno è ineliminabile.

*Osservazione 2*: La semantica può essere vista come una funzione

**Semantica** : **Programmi** -> (**Dati** U {⊥})^**Dati**

L'immagine di tale funzione è particolarmente interessante: rappresenta infatti l'insieme di funzioni sintetizzabili nel linguaggio di programmazione, quelle cioè che ammettono programmi in grado di computarle. Poiché, **Programmi** ha la cardinalità del numerabile, mentre (**Dati** U {⊥})^**Dati** quella del continuo, tale immagine è un sottoinsieme proprio del codominio: esistono cioè funzioni **non computabili** nel nostro sistema di programmazione.
Vedremo in seguito che tale fenomeno è del tutto generale; non solo, ma le classi di funzioni computabili nei vari formalismi vengono a coincidere con la classe delle funzioni ricorsive parziali, che definiremo in seguito.

## 5. LINGUAGGI WHILE

Il linguaggio RAM (ridotto) è in sostanza un assembly semplicissimo, e mal si presta come linguaggio ad alto livello. In particolare, l'istruzione di salto condizionato rende scarsamente leggibili e documentabili i programmi. Introduciamo ora un linguaggio, pensabile come un frammento semplificato di un linguaggio procedurale ALGOL-like, che usa strutture di controllo "un ingresso, una uscita".
Nostro primo obiettivo sarà il confronto della potenza computazionale dei due linguaggi: dimostreremo che essi computano la stessa classe di funzioni. Diamo ora la sintassi del linguaggio while.

**SINTASSI**

- Un **comando di assegnamento** è del tipo:

  $x_k := 0, \quad x_k := x_j + 1, \quad x_k := x_j - 1 \qquad (0 \leq k, j \leq 20)$

  dove $x_k$, $x_j$ sono dette variabili

- Un **comando while** è del tipo:

while $x_k \neq 0$ do $C$ $\quad (0 \leq k \leq 20)$
essendo $C$ un comando arbitrario

- Un **comando composto** è del tipo:
  begin $C_1$; $C_2$; ...; $C_m$ end $\quad (0 < m)$
  essendo $C_1$, ..., $C_m$ comandi arbitrari

- Un **programma while** è un comando composto

Più formalmente, il linguaggio while è generato dalla seguente grammatica:

**GRAMMATICA**:

<prog> -> <com comp>

<com> -> <ass> / <com w> / <com comp>

<ass> -> $<x_k := 0 / x_k := x_{j+1} / x_k := x_j - 1$
$(0 \leq k, j \leq 20)$

<com w> -> while $x_k \neq 0$ do <com>
$(0 \leq k \leq 20)$

<seq com> -> <com> / <com> <seq com>

<com comp> -> begin <seq com> end

La sintassi fornisce una definizione induttiva dell'insieme dei programmi del linguaggio while. Questo permette di dare uno schema di dimostrazione per induzione della proprietà dell'insieme dei programmi. In particolare, perché, una proprietà $P$ valga per tutti i programmi basterà dimostrare (1) e (2), dove:

(1) $P$ vale per tutti i comandi di assegnamento
(2) Se $P$ vale per $C$, allora vale per.;
$C' \equiv$ while $x_k \neq 0$ do $C$
Se $P$ vale per $C_1$, ..., $C_m$, allora vale per
$C' \equiv$ begin $C_1$;...; $C_m$ end

Per lo stesso motivo, risulterà molto utile la definizione ricorsiva per funzioni aventi come dominio l'insieme dei programmi while.

Per la semantica intuitiva del linguaggio while, basta pensarlo come un frammento del PASCAL, studiato in TAMC. Diamo invece più formalmente una semantica operazionale:

**SEMANTICA OPERAZIONALE**:

*Def.1*: **w-Stato** è una funzione $\underline{x}$: {$x_0$, $x_1$, ..., $x_{20}$} -> *N*

A differenza dell'analoga nozione per il linguaggio RAM, non abbiamo necessità di introdurre il registro *L*, in quanto il nostro linguaggio while non ha istruzioni di salto. La limitazione a soli 21 registri di memoria e' per semplicità e, come vedremo, non inficia la capacità computazionale. Indichiamo con **w-Stati** l'insieme degli stati N^21.

*Def.2*: **Dato** è un qualsiasi intero non negativo.

*Def.3*: **Inizializzazione** è la funzione **w-in**: **Dati** -> **w-Stati**, dove

**w-in**(z) = (0, x, 0, 0, ..., 0)

Indichiamo ora con **Comandi** l'insieme dei comandi del linguaggio while. Il punto cruciale è la definizione della semantica dei comandi:

*Def.4*: **Semantica** dei comandi è una funzione parziale:

[[ ]] () : Comandi x **w-Stati** -> **w-Stati**

definita come segue:

Sia *C* un comando, $\underline{x}$ uno stato; definiamo $\underline{y}$ = [[C]](x) (intuitivamente, il risultato del comando *C* sullo stato $\underline{x}$) per ricorrenza.

Caso *C* ≡ $x_k$ := 0  $y_j$ = Se k = j allora 0
altrimenti $x_j$

Caso *C* ≡ $x_k$ := $x_i$ + 1  $y_j$ = Se k = j allora $x_i$ + 1
altrimenti $x_j$

.  .

Caso *C* ≡ $x_k$ := $x_i$ - 1  $y_j$ = Se k = j allora $x_i$ - 1
altrimenti $x_j$

Caso *C* ≡ begin $C_1$;...; $C_m$ end:

[[*C*]] ($\underline{x}$) = [[$C_m$]] o [[$C_{m-1}$]] o...o [[$C_1$]]($\underline{x}$)
essendo o la composizione fra funzioni

Caso *C* ≡ while $x_k \neq 0$ do $C_1$:

$$[[C]](\underline{x}) = \begin{cases} [[C_1]]^{\wedge}(\mu e \text{ (componente k-ma in } [[C_1]]^{\wedge}e\ (\underline{x})=0)\ (\underline{x}) \\ \text{o } \perp \end{cases}$$

dove "μe..." significa "più piccolo intero e tale che..." e

[[$C_1$]]^e = [[$C_1$]] o [[$C_1$]] o ... o [[$C_1$]]  (e volte)

Con μe(...) intendiamo il primo intero non negativo tale che la k-ma

componente della funzione $[[C_1]]^e$ ($[[C_1]]$ composta con se stessa e volte) valutata sull'argomento $\underline{x}$ risulta 0.
Se questo non succede mai, conveniamo di porre $[[C]](\underline{x}) = \perp$
(indefinito) convenendo inoltre che $[[C]] \circ \perp = \perp \circ [[C]] = \perp$.
Siamo ora in grado di associare ad ogni comando *C* una funzione parziale $[[C]]$ : **w-Stati** -> **w-Stati**.
Poiché, un programma *P* è in particolare un comando (comando composto), associamo a *P* la funzione parziale

$[[P]]$ : **w-Stati** -> **w-Stati**

Se, in analogia a quanto visto per il linguaggio RAM, conveniamo di scegliere come risultato della computazione il valore della variabile $x_0$, otteniamo:

*Def.5*: Semantica del programma *P* è la funzione

$\psi_P$: **Dati** -> **Dati** U $\{\perp\}$

dove: $\psi_P(x)$ = Componente di indice 0 in $[[P]]$ (w-in (x)).

Diremo anche che $\psi_P$ è la funzione calcolata dal programma *P*, e $\{\psi_P\}$ il sistema di programmazione del linguaggio while.

Indichiamo con **Programmi** l'insieme dei programmi per RAM (ridotti), con **W-Programmi** l'insieme dei programmi while. Sia:

Funzioni(RAM) = $\{f \mid f$: **Dati** -> **Dati** U $\{\perp\}$,
$\exists\ P \in$ Programmi ($f = \varphi_P$)$\}$

Funzioni(while) = $\{g \mid g$ : **Dati** -> **Dati** U $\{\perp\}$,
$\exists\ P \in$ **W-Programmi** ($g = \psi_P$)$\}$

Affronteremo ora il problema di confrontare queste due classi di funzioni.

## 6. COMPILATORE W-PROGRAMMI --> PROGRAMMI

Siano $L_1$ e $L_2$ due linguaggi di programmazione, con rispettivi sistemi di programmazione $\{\Psi_{P1}\}$ e $\{\varphi_{P2}\}$.

*Def.1*: Una **traduzione** è una funzione $T : L_1 \to L_2$ tale che:

a) *T* è una funzione totale calcolabile in PASCAL
b) $\forall\ P \in L_1\ \ (\Psi_P = \varphi_{T(P)})$

La condizione b) dice che *T* "mantiene" la semantica.
La condizione a) dice che *T* deve essere "effettivamente calcolabile".

Anzi, nella pratica, questa traduzione deve essere effettuata in un tempo proporzionale alla lunghezza del programma sorgente.
Se inoltre $L_1$ è definito induttivamente dalla sintassi, è spesso possibile descrivere la traduzione $T$ in termini ricorsivi sulla costruzione di $L_1$; si parla in tal caso di traduzione guidata dalla sintassi (oggetto del corso di "Linguaggi formali e Compilatori").

Detto **Funzioni** $(L_i)$ l'insieme delle funzioni computabili in $L_i$, è chiaro che se esiste una traduzione $T$: $L_1$ -> $L_2$ allora:

**Funzioni**$(L_1) \subseteq$ **Funzioni**$(L_2)$;

la costruzione di una traduzione (programma compilatore) è allora uno strumento per il nostro problema di raffronto tra classi di funzioni.

Consideriamo ora la classe **Programmi** in versione etichettata: certe istruzioni, cioè, potranno avere una "etichetta" (label) con cui identificarle. La traduzione a programmi in cui l'etichetta è identificata dal numero d'ordine dell'istruzione è diretta.

Esempio:

```
         R1 <- R1 + 1                R1 <- R1 + 1
LOOP    :R0 <- R0 + 1             |-> R0 <- R0 + 1
         IF R2 = 0 GOTO LOOP         IF R2= 0 GOTO 2
```

Costruiamo ora una traduzione

Comp : **W-Programmi** -> **Programmi**

come segue: sia $C$ un qualsiasi comando, allora:

Caso $C \equiv x_k := 0$,

Comp($C$) ≡

```
                .
LOOP   :Rk <- Rk - 1
        IF Rk  = 0  then  goto  EXIT
        IF R21 = 0  then  goto  LOOP
                .
EXIT   :Rk <- Rk - 1
```

Caso $C \equiv x_k := x_j \pm 1$, $(k \neq j)$

Comp($C$) ≡

```
                .
LOOP   :Rk <- Rk - 1
        IF Rk  = 0  then  goto  EXIT 1
        IF R21 = 0  then  goto  LOOP
EXIT 1 :IF Rj  = 0  then  goto  EXIT 2
                .
        Rj  <- Rj  – 1
        R22 <- R22 + 1
        IF R21 = 0  then  goto  EXIT 1
EXIT 2 :IF R22 = 0  then  goto  EXIT 3
                .
        R22 <- R22 - 1
        Rj  <- Rj  + 1
        Rk  <- Rk  + 1
```

IF $R_{21}$ = 0 then goto EXIT 2
.
EXIT 3 : $R_k$ <- $R_k$ ± 1

Caso $C \equiv$ begin $C_1;...;C_m$ end,

$$Comp(C) = \begin{cases} Comp(C_1) \\ Comp(C_2) \\ . \\ . \\ . \\ Comp(C_m) \end{cases}$$

Caso $C \equiv$ while $x_k \neq 0$ do $C_1$,

$$Comp(C) = \begin{cases} LOOP : IF\ R_k = 0\ then\ goto\ EXIT \\ \quad\quad Comp(C_1) \\ \quad\quad IF\ R_{21} = 0\ then\ goto\ LOOP \\ \quad\quad . \\ EXIT : R_{22} <-\ R_{22} - 1 \end{cases}$$

Poiché, i programmi in linguaggio while sono particolari comandi, è chiaro che (restringendone il dominio) Comp è una funzione che ad ogni programma in linguaggio While associa un programma in linguaggio RAM.

**TEOREMA** (Correttezza del compilatore):
Comp: **W-Programmi** -> **Programmi**

è una traduzione.

**Dimostrazione**:

a) Poiché, è definita per ricorsione sullla definizione induttiva di W-Programmi, Comp è una funzione totale; essa è inoltre facilmente programmabile in PASCAL.

b) Dobbiamo ora mostrare che Comp mantiene la semantica, cioè:

$\forall\ P \in$ W-Programmi $(\Psi_P = \varphi_{Comp(P)})$

Cominciamo con l'osservare che se $\overline{C}$ = Comp($C$), allora $\overline{C}$ contiene al più le variabili $R_0$, $R_1$, $R_2$, ..., $R_{20}$, $R_{21}$, $R_{22}$ ($C$ sia un qualsiasi comando).
Con una immediata induzione, si osserva che se i registri $R_{21}$ e $R_{22}$

sono azzerati prima della esecuzione di $\overline{C}$ = Comp($C$), al termine dell'esecuzione il contenuto di $R_{21}$ e $R_{22}$ sarà nuovamente 0.

Associamo ora ad ogni programma $\overline{C}$ = Comp($C$) una funzione:

$$\overline{\delta}_C : N\text{^}21 \text{ -> } N\text{^}21 \text{ U } \{\perp\}$$

che al vettore ($x_0$, $x_1$, ..., $x_{20}$) associa il vettore ($y_0$, $y_1$,...,$y_{20}$) con la seguente regola:

se $x_k$ ( $0 \leq k \leq 20$ ) è il contenuto del registro $R_k$ prima della esecuzione di $\overline{C}$ e i registri $R_{21}$ e $R_{22}$ sono azzerati, allora $y_k$ ($0 \leq k \leq 20$) è il contenuto di $R_k$ dopo la esecuzione di $\overline{C}$ (nel caso in cui la computazione termini).

Poiché **w-Stati** = $N$^21, vale che

$$\delta_{\overline{C}}: \textbf{w-Stati} \rightarrow \textbf{w-Stati} \cup \{\perp\}$$

Dimostriamo ora, utilizzando lo schema di induzione prodotto dalla definizione induttiva dell'insieme dei comandi, che vale

$$[[C]] = \delta_{Comp(C)}$$

$\alpha$) Sia $A$ un comando di assegnamento. La verifica che

$$[[A]] = \delta_{Comp(A)}$$

è diretta

$\beta$) Sia $C \equiv$ begin $C_1;...;C_m$ end: valgono

$$[[C]] \equiv [[C_m]] \text{ o } [[C_{m-1}]] \text{ o } ... \text{ o } [[C_1]] =$$
$$= \delta_{Comp(Cm)} \text{ o } \delta_{Comp(C1)} \quad \text{(Ipotesi d'induzione)}$$

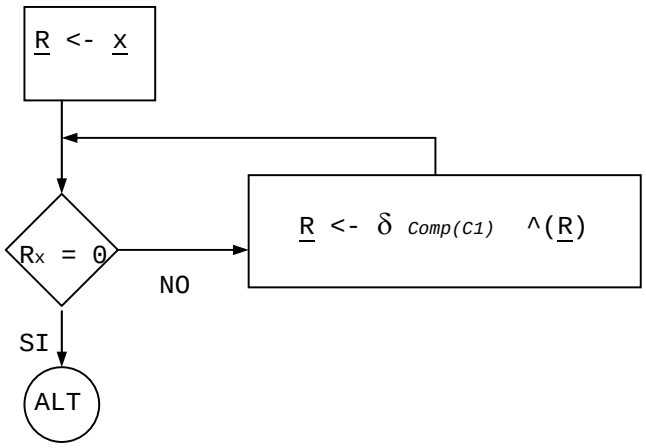
$$\delta_{Comp(C)} = \delta \begin{cases} Comp(C1) = \delta_{Comp(Cm)} \; o \; ... \; o \; \delta_{Comp(C1)} \\ ... \\ Comp(Cm) \end{cases}$$

Quindi:

$$[[C]] = \delta_{Comp(C)}$$

$\gamma$) Sia $C \equiv$ while $x_k \neq 0$ do $C_1$

$\delta_{Comp(C)}(\underline{x})$ è ottenuto dal flow-chart interpretato:




Quindi:

$\delta_{Comp(C)}(\underline{x}) = \delta_{Comp(C1)}\ ^e(\underline{x})$

dove e è il primo intero per cui la k-ma componente di
$\delta_{Comp(C1)}\ ^e(\underline{x})$ è 0

Poiché per l'ipotesi di induzione, $\delta_{Comp(C1)} = [[C_1]]$, vale allora:
$\delta_{Comp(C)}(\underline{x}) = [[C_1]]\ ^e(\underline{x})$,

dove e è il primo intero per cui la k-ma componente è 0.
Poiché inoltre, per definizione della semantica del comando while, è
$[[C]](\underline{x}) = [[C_1]]^e(\underline{x})$

concludiamo:
$[[C]] = \delta_{Comp(C)}$

Abbiamo quindi dimostrato che, per ogni comando *C*, vale:
$[[C]] = \delta_{Comp(C)}$

Sia *P* un programma while; esso è un particolare comando, quindi:
$[[P]] = \delta_{Comp(P)}$

Allora concludiamo:
$\Psi_P(x)$ = O-ma componente di $[[P]]$(w-in x) = O-ma componente di $\delta_{Comp(P)}$ (in x) = $\varphi_{Comp(P)}(x)$.

♦

Una immediata conseguenza del precedente teorema e' che ogni funzione computabile in linguaggio while e' anche computabile in linguaggio RAM (ridotto); vale quindi il seguente:

**TEOREMA**: **Funzioni (while) $\subseteq$ Funzioni (RAM)**

## 7. MACRO ISTRUZIONI

Consideriamo il programma while:

```
begin
          x0 := 0; x14 := x2 +1; x15 := x3 +1;
                                                    .
          while x2 ≠ 0 do begin x0 := x0 +1; x2 := x2 -1 end;
                                                  .
          while x3 ≠ 0 do begin x0 := x0 +1; x3:= x3 -1 end;
                    .              .
          x2 := x14 -1; x3 := x15 -1; x14 := 0; x15 := 0
end
```

Non è difficile sincerarsi che tale programma ha l'effetto principale di trasferire nella variabile $x_0$ la somma dei contenuti delle variabili $x_2$ e $x_3$ (avendo effetti collaterali sulle variabili $x_{14}$, $x_{15}$).
Il programma precedente può essere "abbreviato" scrivendo:

$x_0 := x_2 + x_3$

Quest'ultima scrittura non è definita nel linguaggio di programmazione; possiamo però convenire che essa richiami il comando composto di cui sopra, ed usarla come macroistruzione in un qualsiasi programma while, che non contenga le variabili $x_{14}$, $x_{15}$.
Una più approfondita discussione sulla presentazione delle istruzioni base di alcuni linguaggi di programmazione ad alto livello come macroistruzioni di una variante del linguaggio while è reperibile in "A programming approach to Computability", Kfoury - Arbib.
Ci accontentiamo qui di elencare alcune macroistruzioni utilizzate in seguito. (Scrivere esplicitamente i programmi relativi come esercizio).

**TEOREMA**: Esistono macroistruzioni del tipo

$$x_i := Op(x_j, x_8) \quad (i,j,8 \leq 10)$$

per il seguente elenco di operazioni Op:

- Somma (x + y)
- Prodotto (x . y)
- Sottrazione fra interi positivi (x - y)
- Divisione intera (x / y)
- Modulo ($< x >_y$)
- Coppia di Cantor (< x , y >)

Analogamente, le proiezioni destra (des x) e sinistra (sin x) di Cantor ammettono macroistruzioni del tipo

$$X_j := \text{des } x_8,\ x_j := \text{sin } x_8, \quad (j,8 \leq 10)$$

Vediamo ora come alcuni comandi di linguaggi di programmazione ad alto livello possano essere realizzati con macroistruzioni ($i, j \leq 10$):

if $x_j = 0$ then C ≡

```
begin
   .   .   .
x14 := 1 -(xj –(xj - 1));
                          .
 while x14 ≠ 0 do begin C; x14 := x14 -1 end
end
```

if $x_j = 0$ then $C_1$ else $C_2$ ≡

```
begin
      .   .   .          .
   x14 := 1 -(xj -(xj - 1)); x15 := 1 - x14;
                                  .
 while x14 ≠ 0 do begin C1; x14 := x14 - 1 end
                                .
   while x15 ≠ 0 do begin C2; x15 := x15 - 1 end
end
```

if $x_j \geq x_i$ then $C_1$ else $C_2$ ≡

```
begin
          .
   x14 := xi - xj
 if x14 := 0 then C1 else C2
   x14 := 0
 end
```

Elenchiamo ora una serie di operazioni su interi interpretati come sequenze di interi che ci saranno utili in seguito:
Sia $x = < x_1 <x_2 < ... < x_l, 0 > ... > >$, abbreviata con $(x_1,...,x_l)$.

$Pro(k,x) = $ se $k \leq l$ allora $x_k$
(Proiezione sulla k-ma componente)

$inc(k,x) = (x_1,...,x_k +1,...,x_e)$
(Incrementa la k-ma componente)

.

$dec(k,x) = (x_1,...,x_k -1,...,x_e)$
(Decrementa la k-ma componente)

$lung(x) = l$
(Dà la lunghezza della sequenza)

$ze(l) = (0,0,...,0)$
(Al numero l associa il numero che rappresenta una sequenza di l zeri)

E' di immediata verifica il seguente:

**TEOREMA**: Esistono macroistruzioni del tipo $x_i := Op(x_j,x_8)$ o $x_i := Op(x_j)$ $(i,j,8 \leq 10)$ per tutte le operazioni sopraindicate.

## 8. ARITMETIZZAZIONE DI PROGRAMMI

L'insieme **Programmi** è numerabile, può cioè essere messo in corrispondenza biunivoca con N+ = {1, 2, ..., n, ... }. L'esibizione di una tale corrispondenza ha molte analogie con il concetto di ascissa in geometria analitica: come un punto su una retta con sistema cartesiano può essere visto come numero reale, cosi' un programma può essere pensato come un numero intero. Esibire tale corrispondenza in modo effettivo viene detto aritmetizzare (o godelizzare, dal nome del logico Godel che inaugurò tali tecniche nella metamatematica (1930).

*Def.1*: **Aritmetizzazione** di **Programmi** è una funzione
Cod : **Programmi** -> *N+*

tale che:

(1) Cod è una corrispondenza biunivoca

(2) Cod può essere computata con un programma PASCAL.

Diamo ora una semplice aritmetizzazione di **Programmi**, utilizzando le funzioni $< x, y >$ (Funzione coppia di Cantor), $< x >_3$ (resto della divisione tra x e 3), x/3 (Quoto della divisione tra x e 3):

1) Sia **Istruzioni** l'insieme delle istruzioni. Sia

Ar: **Istruzioni** -> *N*

definito da:

Caso istr = $R_k$ <- $R_k$ +1, Ar(istr) = 3k

.

Caso istr = $R_k$ <- $R_k$ -1, Ar(istr) = 3k + 1

Caso istr = IF $R_k$ = 0 THEN GOTO m,

Ar(istr) = 3. < k, m > -1

Ar è una corrispondenza biunivoca (scrivere per esercizio esplicitamente i programmi per realizzare Ar e Ar^(-1).

2) Cod : **Programmi** -> *N*+ è realizzata come segue: se

Cod(*P*) = $istr_1$; ...; $istr_s$

allora

Cod(*P*) = < Ar($istr_1$), < ... < Ar($istr_s$), 0 > ... >>

Cod è allora una aritmetizzazione (scrivere esplicitamente i Programmi per realizzare Cod e Cod^(-1).

Se j = Cod(*P*), diciamo che j è il numero di Godel di *P*, o che *P* è lo j-mo programma nella numerazione di Godel. In luogo del sistema di programmazione $\{\varphi_P\}$ potremo scrivere $\{\varphi_j \mid j \geq 1\}$
$\varphi_j(x)$ è il risultato della computazione sul dato x del programma il cui numero di Godel è j (o, più suggestivamente, del programma j).

*Esercizi*: Determinare il primo j per cui: $\forall$ x ($\varphi_j$ (x) = $\perp$)
Determinare infiniti $j_k$ per cui: $\forall$ x ($\varphi_{j_k}$ (x) = 0)
E' la funzione $f(x) = \varphi_x(1)$ "intuitivamente" calcolabile?

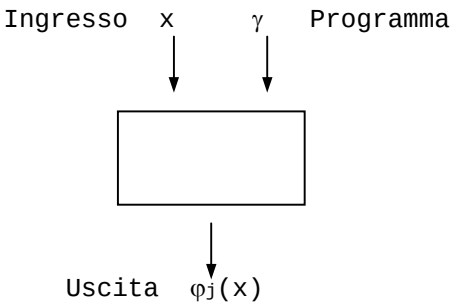
## 9. PROGRAMMA INTERPRETE E FUNZIONE UNIVERSALE

In precedenza abbiamo considerato l'aritmetizzazione

Cod : **Programmi** -> *N*+

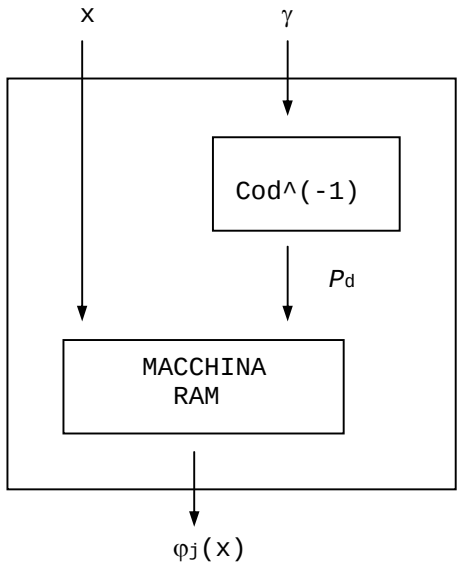
che ci permetteva di interpretare un intero positivo come un programma RAM. Questo ci portava suggestivamente a scrivere il sistema di programmazione $\{\varphi_j\}$, dove $\varphi_j$ (x) è il risultato della computazione sul dato x del programma Cod^(-1)(j).

In altre parole, si considera *N*+ come un linguaggio di programmazione, ed in linea di principio è pensabile la realizzazione di una macchina del tipo:



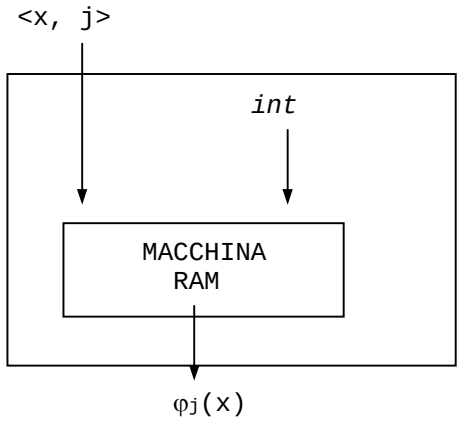

Utilizzando il modello di RAM, vediamo due soluzioni:

**Soluzione 1:**




Cod^(-1): *N*+ -> **Programmi** realizza una traduzione. E' cioè pensabile come un compilatore dal linguaggio "ad alto livello" (!) *N*+ nel linguaggio **Programmi**, accessibile alla macchina.

**Soluzione 2:**



In questo caso la macchina "gira" con un unico programma *int* $\in$ **Programmi** che, accettando in ingresso la coppia < x, j >, dà in uscita $\varphi_j(x)$.

Naturalmente del programma *int* (che chiameremo "interprete") deve essere dimostrata l'esistenza, o, meglio ancora, esso deve essere costruito.
Prima di dettagliare la costruzione di *int*, osserviamo che la soluzione 1 è preferibile se il nostro obiettivo è "far girare" lo stesso programma j, su tanti dati di ingresso $x_1$, $x_2$, ..., $x_n$. Infatti:

"Tempo di computazione" = "Tempo di compilazione di j" +
$$+ \sum_{k=1}^{n} \text{"Tempo di esecuzione sul dato } x_k\text{"}.$$
(Se n è grande, "Tempo di compilazione" è una frazione trascurabile).
L'altra soluzione è ragionevole quando l'obiettivo diventa quello di chiedere" la risposta dei programmi $j_k$ sui dati $x_k$ ($1 \leq k \leq h$) (tipicamente in interattivo).
Dettagliamo ora la costruzione di *int*, utilizzando macroistruzioni del linguaggio while, quindi, sostanzialmente, in linguaggio while.
Per l'esistenza di una traduzione

Comp : **W-Programmi** -> **Programmi**

concluderemo che int è esprimibile in linguaggio RAM.
Nostro obiettivo è di simulare con un programma while l'esecuzione di una RAM con programma $P_j$, di numero di Godel j, sul dato x.
Lo stato di una macchina è definito dal contenuto del registro L e dai contenuti degli (infiniti) registri $R_k$ ($k \geq 0$). Poiché, il programma $P_j$ non contiene esplicitamente la variabile $R_k$, se $k \geq j$, tali variabili permarranno azzerate nell'esecuzione del programma. Potremo pertanto definire, relativamente all'esecuzione di $P_j$, lo stato attraverso il contenuto dei registri L ed $R_k$ ($0 \leq k \leq j+2$)

Diamo il significato in relazione alla simulazione, dei contenuti delle variabili del programma *int*:

$x_0$ contiene ($x_0$, $x_1$, ..., $x_{j+1}$, $x_{j+2}$), cioè lo stato di memoria
($x_k$ va interpretato come contenuto corrente del registro $R_k$ della macchina da simulare)

$x_1$ contiene l, dove l è interpretato come contenuto corrente del registro L.

$x_2$ contiene il dato x

$x_3$ contiene il codice j del programma $P_j$

$x_4$ contiene (eventualmente) il codice dell'istruzione corrente da eseguire sulla macchina da simulare.

**PROGRAMMA** *int* (ingresso <x, y>)

Mette il dato x e il codice programma j rispettivamente nei registri $x_2$, $x_3$

- $x_2$ := sin($x_1$)
- $x_3$ := des($x_1$)

Il contenuto di $x_0$ è posto essere (0, x, 0, 0, ..., 0), simulando l'inizializzazione per lo stato di memoria

- $x_0$ := Ze($x_3$)
- $x_0$ := < 0 < $x_3$, $x_0$ >>

Simula l'inizializzazione di L -> $x_1$ := 1

while $x_1 \neq 0$ do

Finché non si arriva alla condizione di terminazione (L = 0, sse $x_1$ = 0) si desume dal contenuto di $x_1$ l'istruzione corrente da simulare, se esiste (cioè se $x_1 \leq$ lung(j)).
In tal caso se ne mette il codice in $x_4$
Si decodifica e si simula l'istruzione nei due effetti di cambiamento di stato di memoria e di L

if $x_1$ > lung(j) then $x_1$ := 0

else

$x_4$ := Pro($x_1$,$x_3$)

if <$x_4$>$_3$ = 0 then
- $x_2$ := $x_4$/3
- $x_0$ := inc($x_2$,$x_0$)
- $x_1$ := $x_1$ +1

if <$x_4$>$_3$ = 1 then
- $x_2$ := $x_4$/3
- $x_0$ := dec($x_2$,$x_0$)
- $x_1$ := $x_1$ +1

if <$x_4$>$_3$ = 2 then
- $x_5$ := $x_4$/3 +1
- $x_6$ := des($x_5$)
- $x_7$ := sin($x_5$)
- $x_8$ := Pro($x_7$,$x_0$)
- then $x_1$ := $x_3$
- if $x_8$ = 0 else $x_1$ :=$x_1$+1

Si lascia in $x_0$ il

contenuto finale di $R_0$ $x_0 := \sin(x_0)$
nella macchina simulata

Il programma int è scritto in linguaggio while con macroistruzioni. Poiché, esso utilizza esplicitamente 9 variabili ($x_0$, $x_1$, ..., $x_8$) e le macroistruzioni possono essere sostituite con istruzioni usando (facile verifica) al più 5 ulteriori variabili, concludiamo che esiste un programma $int_1$, scritto in linguaggio while, che utilizza 14 variabili e tale che:

$$\psi_{int1}(< x, y >) = \varphi_y(x)$$

Detto $int_2$ il programma RAM tale che $int_2 = Comp(int_1)$, vale che:

$\alpha$) $int_2$ contiene al più 16 variabili $R_k$.

$\beta$) $\varphi_{int2}(< x, y >) = \varphi_{Comp(int1)}(< x, y >) = \varphi_y(x)$

Possiamo allora concludere, ponendo $u = Cod(int_2)$:

**TEOREMA**: Dato il sistema di programmazione RAM $\{\varphi_j\}$, esiste un indice u tale che: $\forall x \forall y (\varphi_u(< x, y >) = \varphi_y(x)$. La funzione parziale $\varphi_u$ è detta funzione universale per $\{\varphi_j\}$.

## 10. ELIMINAZIONE DEL “GOTO”

Abbiamo in precedenza (utilizzando tecniche di compilazione) mostrato che per ogni programma while c'è un programma RAM che computa la stessa funzione.
Chiediamoci ora se, dato un programma RAM, esiste un programma while che computa la stessa funzione.
In termini più generali, il problema diventa: l'eliminazione di istruzioni di salto (nel nostro caso: IF $x_k = 0$ THEN GOTO n) a favore di comandi strutturati (nel nostro caso i comandi composti e il comando while) diminuisce la capacità computazionale? Il fatto che (anche con semplici primitive) la capacità computazionale resti inalterata, costituisce il nucleo dei risultati di Bohm-Jacopini, che giustificano, (rispetto alla potenza computazionale) l'uso di istruzioni strutturate.

**TEOREMA**: Per ogni programma RAM *P* esiste un programma while *W* contenente al più 14 variabili $x_k$ tale che:

$$\forall x (\varphi_P(x) = \varphi_W(x))$$

**Dimostrazione**: Sia $j = Cod(P)$ il numero di Godel di *P*. Consideriamo il seguente programma while con macroistruzioni:

begin $x_0 := j$; $x_1 := <x_1, x_0>$; $x_0 := 0$; $x_0 := \varphi_{int1}(x_1)$ end

dove:

$\alpha$) La macroistruzione $x_0 := j$ sta per

$$\text{begin } \underbrace{x_0 := x_0 + 1;\ x_0 := x_0 + 1;\ \ldots\ ;\ x_0 := x_0 + 1}_{j \text{ volte}} \text{ end}$$

$\beta$) La macroistruzione $x_0 := \varphi_{int1}(x_1)$ è realizzata dal programma $int_1$.

Poiché, $int_1$ usa 14 variabili (compresa $x_0$) e su $x_0$ non ci sono effetti collaterali, il precedente programma abbrevia un programma While *W* che utilizza 14 variabili. Inoltre:

$$\psi_W(x) = \psi_{int1}(< x, j>) = \varphi_j(x) = \varphi_P(x) \qquad \blacklozenge$$

Una parafrasi del risultato precedente è il seguente:

**TEOREMA**: *Funzioni* (RAM) $\subseteq$ *Funzioni* (while)

Una ulteriore conseguenza è il fatto che, utilizzando al più 14 variabili, possiamo simulare qualsiasi programma PASCAL. Ciò giustifica le nostre limitazioni al numero di variabili per *W-**Programmi***.

## 11. LE FUNZIONI RICORSIVE PARZIALI

Abbiamo in precedenza dimostrato che

*Funzioni* (while) = *Funzioni* (RAM) $\not\subset$ (N U {$\perp$})^N

e tratteggiato il fatto che Funzioni (PASCAL) = Funzioni (while). Le classi di funzioni definibili nei vari formalismi esaminati vengono perciò a coincidere. Daremo ora una caratterizzazione algebrica di tale classe.
Consideriamo l'insieme delle funzioni parziali a argomenti interi in interi. Introduciamo degli operatori parziali su tale classe:

*Composizione*: Se $h(x_1, \ldots, x_m)$ è una funzione a m argomenti e $g_1(\underline{x}),\ldots,g_m(\underline{x})$ sono funzioni a n argomenti, la composizione di *h* e $g_1,\ldots,g_m$ (COMP($h$; $g_1,\ldots,g_m$)) è la funzione *f* in n argomenti definita:

$$f(\underline{x}) = h(g_1(\underline{x}), g_2(\underline{x}), \ldots, g_m(\underline{x}))$$

*Ricorsione primitiva*: Se $h(y, z, \underline{x})$ è una funzione a n+2 argomenti e $g(\underline{x})$ una funzione a n argomenti, la ricorsione primitiva applicata a *h* e *g* (RIC PRIM ($h$; $g$)) dà la funzione *f* in n+1 argomenti definita:

$$f(y, \underline{x}) = \text{se } y = 0 \text{ allora } g(\underline{x}) \text{ altrimenti } h(f(y\text{-}1, \underline{x}), y\text{-}1, \underline{x})$$

*Minimalizzazione*: Se $g(y, \underline{x})$ è una funzione in n+1 argomenti, la minimalizzazione (ML($g$)) applicata a $g$ dà luogo alla funzione $f$ in n argomenti dove:

$$f(\underline{x}) = \begin{cases} y \text{ se } g(y, \underline{x}) = 0 \text{ e } y < y \rightarrow g(y, \underline{x}) \text{ definita, } g(y, \underline{x}) \neq 0 \\ \perp \text{ altrimenti} \end{cases}$$

Si usa anche scrivere $f(\underline{x}) = \mu y(g(y, \underline{x}) = 0$ dove $\mu y(..)$ significa "più piccolo y tale che ...".

Consideriamo ora un insieme di semplici funzioni, che chiameremo funzioni base:

*Funzioni base*:

$0^n(x_1, x_2, ..., x_n) = 0$; $S(x) = x+1$; $Pro_k{}^n(x_1,..., x_k,..., x_n) = x_k$

Possiamo ora definire la classe delle funzioni ricorsive parziali come segue:

*Def.1.*: La classe delle funzioni ricorsive parziali $P$ è la più piccola classe contenente le funzioni base e chiusa rispetto a composizione, ricorsione primitiva e minimalizzazione.

La definizione precedente ci consente il seguente schema di induzione per dimostrare proprietà di $P$:

Perché, una proprietà $P$ valga per tutte le funzioni $f \in P$ basta dimostrare (1), (2):

(1) $P$ vale per tutte le funzioni base

(2) $\alpha$) Se $P$ vale per $h$, $g_1$, ..., $g_m$, allora vale per la composizione

$\beta$) Se $P$ vale per $h$, $g$, allora vale per la funzione definita per ricorsione primitiva

$\gamma$) Se $P$ vale per $g$, allora vale per la funzione definita per minimalizzazione.

In seguito, indicheremo con $T$ la classe di funzioni totali $f \in P$, chiamandola classe delle funzioni ricorsive totali. Indicheremo con $P^n$ ($T^n$) la classe delle funzioni ricorsive parziali (totali) in n argomenti.

## 12. FUNZIONI RICORSIVE PARZIALI E PROGRAMMI WHILE

Abbiamo introdotto la classe delle funzioni ricorsive parziali *P* e il sistema di programmazione $\{\psi_P\}$ associato al linguaggio while; la classe di funzioni computabili in questo linguaggio è *Funzioni* (while).
Nostro obiettivo è il confronto tra la classe *P* e *Funzioni* (while).

Il primo problema che sorge è che le funzioni $\{\psi_P\}$ sono a un solo argomento. Per ovviare a questo inconveniente, basta introdurre nel nostro linguaggio il tipo di dati "vettore di dimensione n a componenti intere", rappresentandolo con interi.

Una soluzione è:

$[x_1] = x_1$
$[x_1, x_2] = < x_1, x_2 > - 1$
([,] realizza un isomorfismo [,]: N * N -> N)
$[x_1, x_2, x_3] = [x_1, [x_2, x_3]]$
([,,] realizza un isomorfismo [,,]: N * N * N -> N)
.....

Definiamo le funzioni computabili, da programmi while, a n argomenti come il sistema $\{\psi_P\ ^n\}$, dove $\psi_P\ ^n = \psi_P\ ([x_1,..., x_n])$. In sostanza, l'indice n in alto funziona come una dichiarazione di tipo della variabile in ingresso.

**TEOREMA**: Ogni funzione $f \in P$ è computabile da programmi while.

**Dimostrazione**:

(1)Le funzioni base lo sono, con immediata verifica.

(2) $\alpha$) Sia $h\ (x_1,...,x_m) = \psi_W\ ^m$ per un programma while *W*, e siano $g_k(\underline{x}) = \psi_{B_k}\ ^n$ per programmi while $B_k$ $(1 \leq k \leq m)$. Il seguente programma calcola allora la composizione $h(g_1,...,g_m)$:

```
begin
   x0 := ψB_m (x1);
   x0 := [x0, ψB_m-1 (x1)];
   .
   .
   .
   x0 := [x0, ψB1 (x1)];
   x0 := ψW (x0)
end
```

$\beta$) Sia $h(y, z, \underline{x}) = \psi_W\ ^{(n+2)}$ per un programma while *W* e sia $g(\underline{x}) = \psi_B\ ^n$ per un programma while B. Allora la funzione ottenuta per ricorsione primitiva da *h* e *g* è calcolata dal programma:

```
begin
```

```
   x4 = Pro1(x1);
   x1 = des(x1)
   x0 = ψB (x1)

   while x3 ≠ x4 do   x0: = ψW ([x0, x3, x1])
                      x3: = x3 +1
end
```

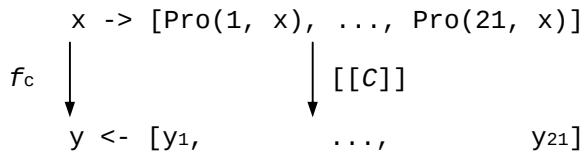
Si noti che il programma è iterativo.

γ) Supponiamo che esista un programma *W* che calcola g, cioè g = ψW ^(n+1). Allora il seguente programma while calcola la funzione f(x) ottenuta applicando a g l'operatore di minimalizzazione:

```
begin
  x2 := ψW ([0, x1])

  while x2 ≠ 0 do    x0 := x0 +1
                     x2:= ψW ([x0, x1])
end                                                    ♦
```

Poniamoci ora il problema opposto: sono le funzioni computate da programmi while ricorsive parziali? Per semplificare la notazione, ricordiamo che ad ogni comando *C* è associata una funzione parziale [[*C*]]: N^21 -> N^21. D'altro lato, usando le funzioni coppia [,], abbiamo mostrato che c'è un isomorfismo N^21 ≈ N  Potremo allora interpretare la semantica [[*C*]] come una funzione parziale $f_C$: N -> N, definita come segue:




1) All'ingresso x associamo il vettore a 21 componenti rappresentato da x.
2) Trasformiamo tale vettore in [$y_1$,..., $y_{21}$] con la funzione [[*C*]].
3) Otteniamo l'uscita y = $f_C$(x) interpretando tale vettore come numero.

Se P è un programma, $\psi_P(x)$ = Pro(1, $f_P$([0, x, 0, ..., 0]). Se dimostriamo che $f_P$ ricorsiva parziale, allora anche $\psi_P$ lo è, in quanto ottenuta per composizione di funzioni ricorsive parziali. Ora, il seguente teorema dimostra che $f_P$ è ricorsiva parziale:

**TEOREMA** Se *C* è un comando del linguaggio while, allora $f_C \in P$.

**Dimostrazione**:

(1) Se A è un assegnamento, allora $f_A \in P$ con verifica immediata.
(2) $\alpha$) Sia $C \equiv$ begin $C_1$, ..., $C_m$ end un comando, e $f_{C_k} \in P$ $(1 \leq k \leq m)$. Allora $f_C = f_{C_m} \circ \ldots \circ f_{C_1} \in P$, in quanto ottenuto per composizione di funzioni di $P$.
$\beta$) Sia $C \equiv$ while $x_k \neq 0$ do $C_1$, e sia $f_{C_1} \in P$. Ricordando la semantica,

$$f_C(x) = f_{C_1}^{(\mu e)(x)}(Pro(k, f_{C_1}^{(e)(x)}) = 0)$$

Consideriamo la funzione $S(y, x) = f_{C_1}^{y}(x)$. Dall'identità

$$f_{C_1}^{y} = f_{C_1}^{1} \circ f_{C_1}^{(y-1)} \quad (y > 0)$$

segue:

$$S(y, x) = \text{se } y = 0 \text{ allora } x \text{ altrimenti } f_{C_1}^{1}(S(y-1, x))$$

$S(y, x) \in P^2$, in quanto definita per ricorsione primitiva di funzioni in $P$. Allora $Pro(k, S(y, z)) \in P$, perché, composizione di funzioni di $P$, e $e(x) = \mu y(Pro(k, S(y, x)) = 0) \in P$, in quanto definita per minimalizzazione di una funzione in $P$.
$S(e(x), x) \in P$, perché, definita come composizione di funzione in $P$, e quindi $f_C(x) \in P$, poiché, $f_C(x) = S(e(x), x)$.

♦

## 13. TESI DI CHURCH

Possiamo a questo punto concludere:

**TEOREMA**: *Funzioni* (RAM) = *Funzioni* (while) = *Funzioni* (PASCAL) = *P*
Abbiamo introdotto diversi formalismi di computazione, e abbiamo dimostrato che invariabilmente questi formalismi permettevano il calcolo della stessa classe di funzioni: le funzioni ricorsive parziali.

Altri formalismi portano ad un analogo risultato, a patto che gli argomenti e i risultati siano opportunamente codificati nelle strutture dati in cui i vari formalismi operano. Per questo è possibile parlare di un concetto di "calcolabilità" astratto, indipendente dal particolare formalismo: calcolabile in PASCAL significa calcolabile in RAM, significa ricorsivo parziale. Possiamo a questo punto enunciare la seguente tesi, che dà un contenuto tecnico al concetto intuitivo di calcolabilità:

**TESI di CHURCH**: La classe delle funzioni intuitivamente calcolabili" è la casse delle funzioni ricorsive parziali.

# 14. ESISTENZA DI PROBLEMI NON DECIDIBILI

Consideriamo problemi esprimibili nella forma: verifica l'elemento x una condizione p?
Più formalmente, descriveremo un problema come una coppia

<<istanza>, <questione>>

dove <istanza> è una variabile su un dato dominio D, <questione> un predicato su D; scriveremo:

Problema: <Nome>
Istanza: x ∈ D
Questione: è la condizione p verificata su x?

Diremo che un problema M è decidibile se esiste una funzione ricorsiva totale : ϕD -> {0,1} (quindi programmabile in un linguaggio di programmazione tipo PASCAL) per cui:

φ(x) = se la condizione p è verificata su x allora 1
altrimenti 0

Richiediamo cioè che la funzione caratteristica dell'estensione di p sia ricorsiva totale:

Esempio 1:
Problema: Equazioni Diofantee Lineari (EDL)
Istanza: a, b, c î N+
Questione: ∃ x ∃ y ( x ∈ Z, y ∈ Z, ax + by = c)?

Esempio 2:
Problema: Equazione per Fermat (EF)
Istanza: n ∈ N+
Questione: ∃ x ∃ y ∃ z (x, y, z ∈ N+, x^n + y^n = z^n)?

Sia $\phi_{EDL}$ la funzione associata al problema esposto in esempio 1: EDL è decidibile, in quanto, come è noto, il seguente programma calcola $\phi_{EDL}$:

```
Ingresso (a, b, c)
      X := MASS. COM. DIV. (a, b, c)
      A := A/X; B := B/X; C := C/X;
      Y := MASS. COM. DIV. (A, B)
      if Y = 1  then z=1
                else z=0
 Uscita (z)
```

Sia ora $\phi_{EF}$ la funzione associata al problema esposto in esempio 2. Si sa che $\phi_{EF}$ (1) = 1, $\phi_{EF}$ (2) = 1, $\phi_{EF}$ (3) = 0, $\phi_{EF}$ (4) = 0; si congettura (grande Teorema di Fermat) che $\phi_{EF}$ (x) = 0 per x ≥ 3.
Se valesse il "grande Teorema di Fermat" $\phi_{EF}$ sarebbe banalmente programmabile. Allo stato attuale di conoscenza, possiamo tutt'al più scrivere il seguente programma;

```
Ingresso (n)
    A := 1; X := 1; Y := 1; Z := 1

                                  ⎧ X := sin(A) +1
    while X^n + Y^n ≠ Z^n do      ⎪ Y := sin(X) +1
                                  ⎨ Z := des(Y) +1
                                  ⎩ A := A+1

    Z := 1
Uscita (Z)
```

Se F è la funzione computata dal precedente programma, è chiaramente:

$$F(n) = \text{Se } \phi_{EF}(n) = 1 \text{ allora } 1 \text{ altrimenti } \perp$$

Il problema diventa quindi se è possibile riconoscere l'esistenza di computazioni infinite che sono responsabili del fatto che, per esempio, $F(3) = \perp$. Se questo fosse possibile, potremmo dare immediatamente un algoritmo di soluzione a questo (e a tanti altri) problemi.
Invece, in questo caso, otterremo un risultato negativo: costruiremo cioè un preciso programma *P* in linguaggio RAM per cui il problema dell'arresto non è decidibile.
Sia w il numero di Godel per la funzione universale per il linguaggio RAM costruita in precedenza. Ricordiamo che

$$\varphi_u(\langle x, y\rangle) = \varphi_y(x)$$

Costruiamo, usando macroistruzioni, il seguente programma *P* in linguaggio RAM:

$$P \equiv R_1 \leftarrow \langle R_1, R_1\rangle;\ R_0 \leftarrow \varphi_u(R_1)$$

Osserviamo che $\varphi_P(x) = \varphi_u(\langle x, x\rangle) = \varphi_x(x)$.
Poniamoci il problema di decidere l'arresto dell'esecuzione di *P* sul dato x. Cioè:

Problema: Arresto per *P*(AP)
Istanza: Un intero x
Questione: E' $\varphi_P(x) \neq \perp$?

(Equivalentemente: è la sequenza di esecuzione di P sul dato x finita?)

**TEOREMA**: (AP) non è decidibile.

**Dimostrazione**: Supponiamo per assurdo che (AP) sia decidibile. Esiste allora una funzione ricorsiva totale $\phi_{AP}$ tale che:

$$\phi_{AP}(x) = \text{se } \varphi_P(x) = \perp \text{ allora } 0 \text{ altrimenti } 1$$

Allora anche la seguente funzione è ricorsiva totale:

$$\psi(x) = \text{se } \phi_{AP}(x) = 0 \text{ allora } 0 \text{ altrimenti } 1 + \varphi_P(x)$$

Esiste quindi un programma RAM, con numero di Godel e, che computa $\psi$, cioè: $\forall$ x $\varphi_e$ (x) = $\psi$ (x)
Vale allora la seguente catena:

$\psi$ (e) = $\varphi_e$ (e) = $\varphi_P$ (e) = se $\phi_{AP}$ (e) = 0 allora 0
altrimenti 1 + $\varphi_P$ (e)

$\phi_{AP}$ (e) non può essere 1, altrimenti sarebbe $\varphi_P$ (e) = 1 + $\varphi_P$ (e). Poiché, $\phi_{AP}$ è totale e prende valori in {0,1}, deve essere $\phi_{AP}$ (e) = 0. Se $\phi_{AP}$ (e) = 0, dalla catena precedente segue che $\varphi_P$ (e) = 0, mentre dalla definizione segue che $\varphi_P$ (e) = $\perp$ (assurdo).

♦

## 15. $S_1$ ^1 FUNZIONE PER IL LINGUAGGIO RAM

Abbiamo visto che il sistema di programmazione RAM $\{\varphi_j\}$ ha una funzione universale $\varphi_u$ tale che: $\forall$ x $\forall$ y ($\varphi_u$ (x, y) = $\varphi_y$ (x)). Questo fatto è una caratteristica generale dei linguaggi di programmazione: una qualsiasi aritmetizzazione del PASCAL (o del linguaggio while, o...) ammette una funzione universale.
Un'altra caratteristica dei linguaggi di programmazione è di poter scambiare "in modo automatico" argomenti e parametri: cioè, se ho un programma *P* che mi calcola la funzione *f*(x, y), fissato un valore di y diciamo $\underline{y}$, devo determinare (e riesco a determinare) un nuovo programma $P_{\underline{y}}$ che mi calcola la funzione (a una sola variabile x) f(x, $\underline{y}$). La legge che ad ogni *P* e ad ogni $\underline{y}$ associa $P_{\underline{y}}$ viene chiamata $S_1$ ^1 funzione del linguaggio di programmazione: in generale essa è una funzione ricorsiva totale.
Determiniamo esplicitamente tale funzione per il linguaggio RAM.

**TEOREMA $S_1$ ^1**: Dato il sistema di programmazione RAM $\{\varphi_i\}$, esiste una funzione ricorsiva totale $S_1$ ^1(i, y) tale che

$\forall$ i $\forall$ y $\forall$ x $\varphi_{S_1\,\hat{}\,1}$ (x) = $\varphi_i$ (<x, y>)

**Dimostrazione**: Sia $P_i$ il programma di numero di Godel i, che computa quindi la funzione $\varphi_i$.

Fissato ora y, sia $\underline{P}_{i,y}$ il segmento programma RAM (scritto con macroistruzioni):

y volte

$R_0$ <- $R_0$ + 1; ...; $R_0$ <- $R_0$ + 1; $R_1$ <- <$R_1$, $R_0$>; $R_0$ <- 0; $P_i$

Si osserva immediatamente che la funzione computata da $\underline{P}_{i,y}$ su ingresso x è $\varphi_i$ (<x, y>). Basta allora porre:

$$S_1^1(i, y) = \text{Cod}(\underline{P}_{i,y})$$

Non è quindi difficile fornire un algoritmo che, dati i e y, dà in uscita il valore $S_1^1(i, y)$; $S_1^1$ è allora una funzione ricorsiva totale.

Il risultato precedente (Teorema $S_1^1$) si generalizza facilmente al caso in cui $\underline{x}$ è un vettore di m variabili e $\underline{y}$ un vettore di n variabili:

**TEOREMA $S_n^m$**: Dato il sistema di programmazione RAM $\{\varphi_i\}$ esiste una funzione ricorsiva totale a n +1 variabili $S_n^m(i, \underline{y})$ tale che:

$$\forall i \forall \underline{y} \quad \forall \underline{x} \; \varphi_{S_n^m(i, \underline{y})}(\underline{x}) = \varphi_i(\underline{x}, \underline{y})$$

Il Teorema $S_n^m$, qui dimostrato per il linguaggio RAM, vale anche per altri linguaggi di programmazione (while, PASCAL, ...).

## 16. SISTEMA DI PROGRAMMAZIONE ACCETTABILE

Abbiamo fino ad ora messo in luce alcune proprietà tipiche di dati linguaggi di programmazione (Linguaggio RAM, linguaggio while).
Abbiamo visto, nei sistemi esaminati, forti analogie, prima fra tutte la capacità di computare la stessa classe di funzioni.
Possiamo a questo punto astrarre rispetto a dettagli inessenziali (dal punto di vista della computabilità) e formulare un concetto generale di sistema di programmazione. La definizione che daremo , dovuta a Roger (1958).
Intanto (poiché, a questo livello i problemi di rappresentazione sono poco rilevanti), identificheremo con N il linguaggio di programmazione (pensare all'aritmetizzazione).
I requisiti che richiederemo sono:

1) Un sistema di programmazione deve contenere programmi per tutte le funzioni ricorsive parziali di un argomento ($P^1$).
2) Deve esistere un programma universale u che, ricevuta in ingresso la coppia <x, y>, simula il programma y sul dato x.
3) Deve essere possibile scambiare argomenti e parametri in modo automatico (Teorema $S_n^m$)

Formalmente:

Def.1: Un sistema di programmazione (enumerazione di Godel, classe di macchine) accettabile è una funzione suriettiva $\varphi : N \to P^1$, che

indicheremo con $\{\varphi_i\}$ (dicendo che $\varphi_i(x)$ è la funzione computata dal programma i) tale che

α) $\exists u \forall x \forall y (\varphi_u (< x, y >) = \varphi_y (x))$ (Esistenza della funzione universale).

β) Esiste una funzione ricorsiva totale $S_1^1 (x, y) \in T^2$ tale che $\forall i \forall y \forall x(\varphi_i (< x, y >) = \varphi_{S_1^1(i, y)} (x))$ (Teorema $S_n^m$).

Il concetto di "traduzione" può essere ora definito in termini generali, come morfismo di sistemi di programmazione:

Def.2: Dati due sistemi di programmazione $\{\varphi_i\}$ e $\{\psi_i\}$, una traduzione t da $\{\varphi_i\}$ a $\{\psi_i\}$ è una funzione $t : N \to N$ tale che:

α) t è ricorsiva totale

β) $\forall i \ \psi_{t(i)} = \varphi_i$ ).

Nonostante la generalità degli assiomi definenti il concetto di programmazione, vedremo che è possibile ottenere in modo semplice risultati di profondo significato: in particolare, il teorema di ricorsione e il teorema di isomorfismo.
Il teorema di ricorsione permette di "scrivere" programmi che fanno riferimento a se stessi, mentre il teorema di isomorfismo stabilisce che tutti i sistemi di programmazione sono isomorfi tra loro (nel senso che esiste sempre una traduzione invertibile).

## 17. IL TEOREMA DI RICORSIONE

Consideriamo i seguenti problemi:

(1) E' possibile determinare un programma RAM di indice j tale che:

$\varphi_j$ = Funzione computata da: $R_0 \leftarrow \varphi_j (R_1)$; $R_0 \leftarrow R_0 - 1$

(2) E' possibile determinare un programma RAM di indice j che "stampa se stesso", cioè tale che:

$$\forall x (\varphi_j (x) = j)$$

(3) I programmi while possono essere considerati (aggiungendo qualche dettaglio) particolari programmi PASCAL, cioè:

$$W\text{-}Programmi \subseteq PASCAL\text{-}Programmi$$

Sia $\psi_P$ la funzione calcolata dal programma PASCAL *P*. Un "compilatore completamente errato" è una funzione ricorsiva totale E: *PASCAL-Programmi -> W-Programmi* tale che:

$\forall P (\psi_{E(P)} \neq \psi_P)$. E' possibile costruire un "compilatore completamente errato"?

(1), (2), (3) possono essere considerati collegati direttamente al seguente problema generale:

Dato un sistema di programmazione $\{\varphi_j\}$ e una funzione ricorsiva totale $f \in T^1$, c'è una soluzione all'equazione:

$$\varphi_{f(j)} = \varphi_j \text{ (incognita j) ?}$$

Analizziamo (1): si tratta di determinare un programma j che usa come macroistruzione la funzione $\varphi_j$ da esso stesso computata.
Sebbene le soluzioni di (1) siano ottenibili direttamente, allo scopo di ridurlo al più generale problema di cui sopra osserviamo che, assegnato j, il numero di Godel f del programma

$$R_0 \leftarrow \varphi_j (R_1);\ R_0 \leftarrow R_0 - 1$$

è facilmente calcolabile con l'algoritmo:

($A_1$) Determinare la sequenza $(j_1, \ldots, j_m) = j$
($\langle j_1,\ldots,j_m\rangle = \langle j_1\langle\ldots\langle j_m, 0\rangle\ldots\rangle$)

(A2) Poni $f = \langle j_1, \ldots, j_m\rangle 1\rangle$ (Infatti Cod ($R_0 \leftarrow R_0 - 1$) = 1)

Abbiamo pertanto costruito una funzione ricorsiva totale f = f(j), e la soluzione al problema (1) è ridotta a risolvere l'equazione $\varphi_j = \varphi_{f(j)}$ (incognita j).

Analizziamo (2): Ad ogni indice j associamo il programma:

j volte

$$R_0 \leftarrow R_0 +1;\ R_0 \leftarrow R_0 +1;\ \ldots;\ R_0 \leftarrow R_0 +1$$

Tale programma computa la funzione che, ad ogni ingresso x, associa sempre j, e il suo numero di Godel z è facilmente ottenibile da j: z = Ze(j). Una soluzione $\underline{j}$ dell'equazione $\varphi_j = \varphi_{Ze(j)}$ risolve il nostro problema, infatti:

$$\forall\ x\ (\varphi_{\underline{j}}\ (x) = \varphi_{Ze(\underline{j})}\ (x) = \underline{j})$$

Analizziamo (3): sia Cod : *PASCAL-Programmi* -> N una aritmetizzazione del PASCAL. La funzione t = t(j) che al codice j del programma sorgente $P_j$ associa il codice f del programma "compilato in modo errato" E($P_j$) è facilmente ottenibile, dato E:

$$t(j) = Cod\ (E\ (Cod\ ^{(-1)}\ (j)))$$

Viceversa, ottenuta t(j), possiamo costruire E come segue:

$$E(Pj) = Cod\ ^{(-1)}\ (t\ (Cod P_j))$$

Il nostro problema è ridotto a determinare una funzione $t \in T^1$ tale che:

1) $\forall$ j (Cod ^(-1) (t(j)) Š un programma while
2) $\forall$ j ($\varphi_j \neq \varphi_{t(j)}$ )

Il teorema di ricorsione stabilisce proprio che l'equazione $\varphi_j = \varphi_{t(j)}$ con t funzione ricorsiva totale, ammette sempre almeno una soluzione j.
Problema (1) ammette quindi soluzione; generalizzando, ciò significa che, per ogni procedura ricorsiva, c'è una procedura iterativa che computa la stessa funzione.

Anche Problema (2) ammette soluzione: esiste sempre un programma, che, su ogni ingresso, stampa il proprio testo.
Problema (3), invece, non ammette mai soluzione, in quanto, comunque scelga t, si riesce sempre a trovare j per cui $\varphi_j = \varphi_{t(j)}$: allora, posto *P* = Cod ^(-1)(j), il programma compilato E(*P*) calcola la stessa funzione di P (E' impossibile scrivere programmi compilatori completamente errati!).

Passiamo ora ai dettagli dimostrativi.

**TEOREMA (di Ricorsione)**: Sia t : N -> N una funzione ricorsiva totale, e sia $\{\varphi_j\}$ un sistema di programmazione. Esiste allora un programma" n $\in$ N tale che: $\varphi_n = \varphi_{t(n)}$.

**Dimostrazione**: Sia u l'indice di una funzione universale per $\{\varphi_j\}$ (Cioè: $\varphi_u (x, y) = \varphi_y (x)$). Vale allora:

$$\varphi_{\varphi_i(i)}(x) = \varphi_{\varphi_u(i,i)}(x) = \varphi_u (x, \varphi_{u(i,i)})$$

Poiché, la composizione di funzioni ricorsive parziali è ricorsiva parziale, $\varphi_u (x, \varphi_{u(i,i)})$ è ricorsiva parziale nelle variabili x, i. Esiste allora un indice e $\in$ N per cui: $\varphi_u (x, \varphi_{u(i,i)}) = \varphi_e(x,i)$. Per il Teorema $S_1$ ^1 vale: $\varphi_e(x,i) = \varphi_{S_1 ^1 (e,i)} (x)$, concludiamo allora:

$$(*)\varphi_{\varphi_i(i)} = \varphi_{S_1 ^1 (e, i)} \qquad \text{(Per un opportuno e)}$$

Poiché, $S_1$ ^1 (e, i) e t(i) sono funzioni ricorsive totali (per il Teorema $S_1$ ^1 e per ipotesi) la loro composizione t($S_1$ ^1 (e, i)) è ricorsiva totale. Esiste allora un indice m per cui:

$$(**)\varphi_m(i) = t(S_1 ^1 (e, i)) \qquad \text{(Per un opportuno m)}$$

Sia n = $S_1$ ^1 (e, m) (n esiste, poiché, $S_1$ ^1 è ricorsiva totale). Vale: $\varphi_n = \varphi_{t(n)}$ Infatti:

$$\varphi_n = \varphi_{S_1 ^1 (e, m)} \qquad \text{(Per definizione di n)}$$

$\varphi S_1 \hat{}1\ (e, m) = \varphi\varphi m(m)$ (Per la (*) )

$\varphi\varphi m(m) = \varphi t(S_1 \hat{}1\ (e,m)$ (Per la (**) )

$\varphi t(S_1 \hat{}1\ (e,m) = \varphi t(n)$ (Per definizione di n)

L'inizio e la fine di questa catena di uguaglianze dimostra pertanto il teorema.

## 18. SEMANTICA DI PROGRAMMI RICORSIVI

## 19. POSET COMPLETI STRETTI

Descriviamo in questo paragrafo i prerequisiti matematici per introdurre la semantica cosiddetta "punto fisso" per i programmi while ricorsivi.
Un insieme parzialmente ordinato (o poset, acronimo delle traduzione inglese) è una coppia $< A, \leq >$, dove A è un insieme e $\leq$ una relazione binaria su A che verifica le proprietà:
$\alpha$) Riflessiva: $\forall\ x\ (x \leq x)$
$\beta$) Transitiva: $\forall\ x\ \forall\ y\ \forall\ z\ (x \leq y \wedge y \leq z \Rightarrow x \leq z)$
$\gamma$) Antisimmetrica: $\forall\ x\ \forall\ y\ (x \leq y \wedge y \leq x \Rightarrow x = y)$

Def.1 Dato un sottoinsieme $S \leq A$, diciamo minimo di S l'elemento m (se esiste) tale che:
$\alpha$) $\forall\ x\ (x \in S \Rightarrow m \leq x)$ $\beta$) $m \in S$

Def.2 Una successione $\{f_k\}$ di elementi di A è detta monotona se
$f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n \leq \ldots$

Def.3 Un maggiorante per $\{f_k\}$ è un qualsiasi $a \in A$ tale che, per ogni k, $a \geq f_k$ il minimo di maggioranti per $\{f_k\}$ (se esiste) è detto limite di $\{f_k\}$; in altre parole, $\lim f_k = l$ significa:

$\alpha$) $l \geq f_k$, per ogni k.
$\beta$) Se, dato a, è $a \geq f_k$ per ogni k, allora $l \leq a$

Possiamo ora definire la struttura che utilizzeremo in seguito.

Def.4 Un poset completo con minimo è un poset $< A, \leq >$ che ammette un minimo (che denoteremo con $\perp$) e tale che ogni successione monotona ammette limite (cioè è completo )

Esempio 1: Un classico esempio è <R+, ≤ >, dove R+ = Insieme dei reali non negativi U{∞}. ≤ è la usuale relazione di ordinamento fra reali, dove inoltre ∞ ≥ x, per ogni x. Il minimo è lo 0, ogni successione non decrescente è notoriamente convergente.

Esempio 2: Questo esempio di poset completo è fondamentale per la nostra semantica. Sia N = {0, 1, 2, ..., n, ...} l'insieme degli interi. Denotiamo con *P*Fun (N, N) l'insieme delle funzioni parziali di N in se stesso. Data una funzione parziale f, il campo di esistenza $Dom_f$ è l'insieme dei valori per cui f è definita.
Definiamo ora la seguente relazione d'ordine ≤ fra funzioni: f ≤ g sse

(1)$Dom_f \subseteq Dom_g$ (2)$x \in Dom_f \Rightarrow g(x) = f(x)$

Il significato intuitivo di f ≤ g è che g è coerente ad f (coincide dove f è definita) ma contiene una quantità maggiore di "informazione" (cioè $Dom_g \geq Dom_f$).
< *P*Fun(N, N), ≤ > è un poset completo con minimo. Infatti:
α) Esiste il minimo, che è la funzione ⊥ ovunque indefinita (cioè $Dom_\perp = \phi$).

β) Data una sequenza monotona $\{f_k\}$, essa ammette sempre limite l. Tale limite, come si può facilmente verificare, è definito come segue:

(1) $Dom_l = U_k\ Domf_k$
(2) Se $x \in Dom_l$, allora esiste k per cui $x \in Domf_k$. Poniamo allora, per tale k: $l(x) = f_k(x)$.

In seguito saremo interessati allo studio dei punti fissi di trasformazioni continue di poset completi con minimo.

Def. 5 Una trasformazione continua T : <A, ≤ > -> < A, ≤ > è una funzione T : A -> A tale che, per ogni successione monotona $f_k$, vale:

$$\lim T(f_k) = T(\lim f_k)$$

E' facile osservare che ogni trasformazione continua T è anche monotona, nel senso che se f ≤ g allora T(f) ≤ T(g).

Def. 6 Punto fisso di una trasformazione T è un elemento x tale che x = T(x).

Il seguente risultato fondamentale mostra che una trasformazione continua di un poset completo con minimo ammette sempre un punto fisso, anzi, ne ammette uno minimo costruibile con tecnica iterativa.

**TEOREMA**: Sia T : < A, ≤ > -> < A, ≤ > una trasformazione continua di < A, ≤ > in se stesso. Allora T ammette un minimo punto fisso $\underline{x} = T(\underline{x})$, dove inoltre: $\underline{x} = \lim t\ ^k(\perp)$.

**Dimostrazione**: Poiché, ⊥ è il minimo, sarà ⊥ ≤ T (⊥).
Poniamo ora

$$\underbrace{T(T(...(T(\perp))...))}_{\text{n volte}} = T^n(\perp)$$

Essendo T continua, essa sarà monotona, quindi:

$$\perp \leq T(\perp) \Rightarrow T(\perp) \leq T^2(\perp) \Rightarrow T^2(\perp) \leq T^3(\perp) \Rightarrow$$
$$\ldots \Rightarrow T^k(\perp) \leq T^{(k+1)}(\perp) \Rightarrow \ldots$$

Pertanto $\perp \leq T(\perp) \leq T^2(\perp) \leq \ldots \leq T^k(\perp) \leq \ldots$, cioè $\{T^k(\perp)\}$ è monotona.
Essendo il poset completo, tale successione ammette limite:

$$\lim \{T^k(\perp)\} = \underline{x}$$

Vale:

$T(\underline{x}) = T(\lim(T^k(\perp))) = \lim T(T^k(\perp))$ (Per la continuità di T)
$= \lim T^{(k+1)}(\perp) = \underline{x}$

Cioè: $\underline{x} = \lim T^k(\perp)$ è un punto fisso di T.
Per mostrare che $\underline{x}$ è il minimo punto fisso, sia x un qualsiasi punto fisso, cioè $x = T(x)$.
E' ovviamente $\perp \leq x$, quindi

$$T(\perp) \leq T(x) = x \ldots, T^n(\perp) \leq T^n(x) = x, \ldots$$

Quindi, per ogni n, è $T^n(\perp) \leq x$, e in conclusione:

$$\underline{x} = \lim T^k(\perp) \leq \lim x = x$$

## 20. PROGRAMMI WHILE RICORSIVI

In questo paragrafo ci poniamo l'obbiettivo di estendere il linguaggio di programmazione while, in modo di poter costruire programmi ricorsivi. Daremo due distinte semantiche (operazionale, punto fisso) dimostrandone l'equivalenza, ed infine proveremo che, per ogni programma ricorsivo, ne esiste uno non ricorsivo che computa la stessa funzione.
Il problema (1) introdotto nello scorso paragrafo chiedeva di determinare una funzione definita da un programma contenente istruzioni riferentesi alla funzione stessa. Questo problema era stato ridotto alla ricerca di soluzioni dell'equazione $\varphi_j = \varphi_{t(j)}$ (dove t funzione ricorsiva totale), e il teorema di ricorsione dimostrava l'esistenza di almeno una soluzione. Tentativamente, saremmo tentati di descrivere la semantica di un programma ricorsivo W in due passi:

(a) Associando al programma W una funzione ricorsiva totale
$t_W : N \to N$ (Disciplina di ricorsione)

(b) Definendo semantica del programma W la funzione $\varphi_j$ tale
che $\varphi_j = \varphi_{t_W(j)}$

Il punto è che l'equazione $\varphi_j = \varphi_{t_W(j)}$ può in generale ammettere soluzioni distinte, e quindi la semantica non sarebbe univoca.
D'altro lato, le trasformazioni $t_W$ sono di tipo abbastanza particolare (come vedremo in seguito) e l'uso del teorema di ricorsione appare troppo generico.

Cominciamo allora definendo sintassi e semantica (operazionale) dei programmi while ricorsivi, vedendo solo in un secondo tempo come il teorema di ricorsione può essere specializzato in questo ambito:

**SINTASSI**: Un programma while ricorsivo è del tipo:

```
Procedura F
      C
```

dove C è un comando composto definito come nella sintassi del linguaggio while, con l'aggiunta che fra i comandi di assegnamento va elencato il comando di tipo: $x_0 := F(x_1)$
Nel seguito, riassumeremo con $F_C$ il programma ricorsivo

```
Procedura F
      C
```

Esempio: I seguenti sono programmi while ricorsivi:

```
Procedura F              Procedura F
Begin                    Begin
  while x1 ≠ 0 do          while x1 ≠ 0 do
      Begin     .              Begin     .
        x1 := x1 -1              x1 := x1 -1
        x0 := x0 +1              x0 := F(x1)
        x0 := x0 +1             end
      end                    end
end
```

**SEMANTICA**: Semantica di un programma ricorsivo $F_C$ è la funzione parziale $\psi_{F_C} : N \to N$ definita come segue:

$\psi_{F_C}(x)$ = Componente di indice 0 in $[[C]]$ (in x)

dove la funzione di inizializzazione sul dato x (in x) è definita come per i programmi while, gli stati sono definiti come per i programmi while, la semantica dei comandi [[ ]] è definita come per i programmi while, tranne che per l'assegnamento di tipo $x_0 := F(x_1)$ (non previsto nei programmi While).
La semantica dell'assegnamento $x_0 := F(x_1)$ dipende dal contesto

```
Procedura F
      C
```

in cui tale assegnamento compare. Useremo quindi la struttura $[[x_0 := F(x_1)]]_C$ per indicare la trasformazione di stato in stato definita dal comando $x_0 := F(x_1)$ nel contesto C. Tale trasformazione è definita come segue:
siano $\underline{x} = (x_0, x_1, ..., x_{20})$ e $\underline{y} = (y_0, y_1, ..., y_{20})$ due stati, allora

$\underline{y} = [[x_0 := F(x_1)]]_C(\underline{x})$ sse $y_k$ = se k = 0 allora $\psi_{F_C}(x_1)$
altrimenti $x_k$

Se $\psi_{F_C}(x_1)$ non è definita, conveniamo di porre

$[[x_0 = F(x_1)]]_C\ (\underline{x}) = \perp$

Esempio: Dimostrare che il seguente programma while ricorsivo C (con macroistruzioni)

Procedura F
if $x_1 = 0$ then $x_0 := 0$

$$\text{else} \begin{cases} x_1 := x_1 - 1 \\ x_0 := F(x_1) \\ x_0 := x_0 + 1 \end{cases}$$

calcola la funzione $\psi_{F_C}(x) = x$.
Per induzione:

a) Se $x = 0$, si verifica immediatamente che $\psi_{F_C}(0) = 0$

b) Supponiamo che $\psi_{F_C}(x) = x$. Dimostriamo che

$$\psi_{F_C}(x+1) = x + 1$$

Infatti:

$$[[\text{if } x_1 = 0 \text{ then } \ldots]]\ (\text{in}(x+1)) =$$
$$= [[\text{if } x_1 = 0 \text{ then } \ldots]]\ (0, x+1, 0, \ldots, 0) =$$
$$= [[ \begin{cases} x_1 := x_1 - 1 \\ \ldots \end{cases} ]] \circ [[x_1 := x_1 - 1]]\ (0, x+1, 0, \ldots, 0) =$$
$$= [[ \begin{cases} x_0 := F(x_1) \\ x_0 := x_0 + 1 \end{cases} ]] \circ [[x_1 := x_1 - 1]]\ (0, x+1, 0, \ldots, 0) =$$
$$= [[ \begin{cases} x_0 := F(x_1) \\ x_0 := x_0 + 1 \end{cases} ]]\ (0, x, 0, \ldots, 0) =$$
$$= [[x_0 := x_0 + 1]] \circ [[x_0 := F(x_1)]]_C\ (0, x, 0, \ldots, 0) =$$
$$= [[x_0 := x_0 + 1]]\ (\psi_{F_C}(x), x, 0, \ldots, 0) =$$
$$= [[x_0 := x_0 + 1]]\ (x, x, 0, \ldots, 0) =$$
$$= (x+1, x, 0, \ldots, 0) =$$

Quindi: $\psi_{F_C}(x+1) = x + 1$.

*Osservazione 1*: nel caso particolare in cui il programma while ricorsivo $F_C$ non contenga l'istruzione $x_0 := F(x_1)$, la funzione computata da $F_C$ coincide con quella computata dal programma while C, cioè: $\psi_{F_C} = \psi_C$.

*Osservazione 2*: la semantica dei programmi while è definita ricorsivamente, ma in termini di cambiamento di stato; per questo diremo che è una semantica operazionale.

**SEMANTICA PUNTO FISSO**:
In questo paragrafo daremo una semantica alternativa ai programmi while ricorsivi, seguendo il seguente schema:

1) Assoceremo ad ogni programma while ricorsivo $F_C$ una trasformazione $T_C$: *P*Fun(N, N) -> *P*Fun(N, N) (dove *P*Fun(N, N) denota l'insieme parzialmente ordinato delle funzioni parziali da N in N)

2) Poiché, si dimostra che $T_C$ è continua, definiremo "semantica di C" il minimo punto fisso di $T_C$ (tale punto fisso, per quanto visto precedentemente, esiste ed è unico).

Sia Procedure F
C
un dato programma while ricorsivo. Fissata una qualsiasi funzione parziale *f*: N -> N, definiamo come segue una semantica $[[\ ]]_f$ per C, dipendente da *f*: stati, inizializzazione, semantica dei vari comandi while sono definiti esattamente come per il linguaggio while; la semantica del comando $x_0 = F(x_1)$ è definita con l'ausilio di *f* dalla seguente:

$$\underline{y} = [[x_0 := F(x_1)]]_f\ (\underline{x}) \text{ sse } y_k = \text{se } k = 0 \text{ allora } f(x_1) \text{ altrimenti } x_k$$

Al solito, definiamo semantica del programma $F_C$ la funzione parziale *g*: N -> N ottenuta da

$$g(x) = \text{Componente di indice 0 in } [[C]]_f\ (\text{in } x)$$

In questo modo, fissato un programma $F_C$, otteniamo una legge che ad ogni funzione parziale $f \in P\text{Fun}(N, N)$ associa una funzione parziale $g \in P\text{Fun}(N, N)$, realizzando così una trasformazione

$$T_C: P\text{Fun}(N, N) \to P\text{Fun}(N, N)$$

definita appunto da $g = T_C(f)$.

Il seguente Teorema è cruciale per la possibilità di definire la semantica "punto fisso";

**TEOREMA**: $T_C$: *P*Fun(N, N) -> *P*Fun(N, N) è una trasformazione continua.

**Dimostrazione (cenni)**: Conviene dimostrare, per induzione sulla costruzione dei programmi, che la trasformazione
[[ ]] : *P*Fun(N, N) -> *P*Fun(N^21, N^21)
è continua.

1) Per i comandi di assegnamento la prova è immediata.
2) Sia $f_1 \le f_2 \ldots \le f_k \le \ldots$, $\lim_{k\to\infty} f_k = f$, allora:

$$\lim_{k\to\infty} [[C_1; C_2]]f_k = \lim_{k\to\infty} [[C_1]]f_k \circ [[C_2]]f_k =$$

$$= \lim_{k\to\infty} [[C_1]]f_k \text{ o } \lim_{k\to\infty} [[C_2]]f_k$$

(Poiché, il limite della composizione è la composizione dei limiti)
$=[[C_1]]_f$ o $[[C_2]]_f$ (Per ipotesi di induzione) $= [[C_1; C_2]]_f$.
Una dimostrazione simile vale per il comando while.
Essendo quindi la trasformazione [[ ]]`f' continua, poiché,:
$T_C(f)(x)$ = Componente di indice 0 in [[ ]] (in (x))

anche $T_C$ risulta continua (in quanto composizione delle funzioni continue inizializzazione, [[ ]], proiezione sulla componente di indice 0).
Ricordando ora che l'ordinamento in $\mathcal{P}$Fun(N, N) dà luogo a un poset completo stretto, è ben posta la seguente nozione:

SEMANTICA PUNTO FISSO: Dato un programma while ricorsivo

Procedura F
C

la semantica di tale programma è il minimo punto fisso $u_C$ della trasformazione (continua) $T_C$.

Ricordiamo che, detta $\perp$ la funzione ovunque indefinita, il minimo punto fisso di $T_C$ si ottiene come $\lim_{k\to\infty} T_C\,\hat{}\,k\,(\perp)$.

Posto $T_C\,\hat{}\,k(\perp) = f_k$, otteniamo la sequenza di funzioni monotona (non decrescente)

$$\perp = f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_k \leq \ldots \leq u_C$$

Intuitivamente, ogni funzione "aggiunge" informazioni alla precedente, approssimando sempre più la funzione "computata", secondo la nostra semantica, $u_C$.

## 21. CONFRONTO FRA LE DUE SEMANTICHE

Abbiamo introdotto due semantiche per programmi while ricorsivi: al programma
Procedura F
C
la semantica operazionale permetteva di associare la funzione $\psi_{F_C}$, quella punto fisso di associare la funzione $\mu_C$.

Obbiettivo di questa sezione è di mostrare l'equivalenza delle due semantiche, cioè che, per ogni C, è $\psi_{F_C} = \mu_C$.
Sia assegnato un programma while ricorsivo
Procedura F
C
e sia $\psi_{F_C}$ la funzione computata secondo la semantica operazionale. Questo equivale a dire che il comando $x_0 = F(x_1)$ in C è interpretato

come la macroistruzione $x_0 := \psi_{Fc}(x_1)$, e quindi, detta $T_c$ la trasformazione associata a C, vale:

$$\psi_{Fc} = T_c(\psi_{Fc})$$

Quindi $\psi_{Fc}$ è punto fisso di $T_c$, ed essendo la semantica d'ordine $\mu_c$ il minimo punto fisso, ne segue che:

$$\mu_c \leq \psi_{Fc}$$

Per concludere che $\mu_c = \psi_{Fc}$ basta allora provare che, se $\psi_{Fc}(x)$ è definita, allora anche $\mu_c(x)$ lo è.
Sia k il numero di volte in cui viene eseguito il comando $x_0 := F(x_1)$ nel calcolo di $\psi_{Fc}(x)$, se $\psi_{Fc}(x)$ è definita (k dipende ovviamente dal dato x).
Tale k sarà detto "numero di chiamata" della procedura sul dato x.
Poniamo ora $f_k = T\,\hat{}\,k(\perp)$ (ricordiamo che $\perp$ è la funzione ovunque indefinita). Dimostriamo ora per induzione il seguente:

Fatto 1: Se $\psi_{Fc}(x)$ è definita con chiamate, allora $\psi_{Fc}(x) = f_{k+1}(x)$.

(1) Sia $k = 0$.
$\psi_{Fc}(x)$ è computata senza nessuna chiamata, cioè il comando $x_0 := F(x_1)$ non viene mai eseguito. Allora il risultato non cambia "interpretando" F come la funzione ovunque indefinita, cioè:
$$\psi_{Fc}(x) = T_c(\perp)(x) = f_{\perp}(x)$$

(2) Supponiamo che la proprietà sia vera per $l \leq k-1$. Dimostriamola vera per k. Sia $\psi_{Fc}(x)$ computata con k chiamate. Se nel corso della computazione deve essere valutata la semantica del comando $X_0 := F(x_1)$, tale valutazione richiede l chiamate della procedura, con $l \leq k-1$.
Allora, per ipotesi di induzione, F può essere interpretata come la funzione $f_{l+1}$, che risulta essere definita sul particolare valore dell'argomento $x_1$ ottenuto dal contenuto del registro $x_1$.
Poiché, $l+1 \leq k$, ne segue che $f_{l+1}(x_1) = f_k(x_1)$.
In altre parole, $\psi_{Fc}$, sul dato x, può essere ottenuta dal programma while che si ottiene sostituendo in C l'istruzione $x_0 := F(x_1)$ con $x_0 := f_k(x_1)$ (con l'ovvia semantica).
Ma allora: $\psi_{Fc}(x) = T_c(f_k)(x) = f_{k+1}(x)$.

Possiamo a questo punto provare il risultato principale:\

**TEOREMA (Equivalenza delle due semantiche)**
Per ogni programma while ricorsivo

Procedura F
C

vale che

$$\psi_{F_C} = \mu_C$$

**Dimostrazione**: Abbiamo visto che $\mu_C \leq \psi_{F_C}$.

Se $\psi_{F_C}(x)$ è definita, essa lo sarà con un numero k di chiamate di procedura, quindi, da Fatto 1:

$$\psi_{F_C}(x) = f_{k+1}(x) = T\ ^{k+1}(\perp)(x)$$

Poiché, $f_{k+1} \leq \mu_C$, essendo $f_{k+1}(x)$ definito, sarà $f_{k+1}(x) = \mu_C(x)$, e quindi

$$\psi_{F_C}(x) = \mu_C(x)$$

Ma allora $\psi_{F_C} \leq \mu_C$ , che permette di concludere: $\psi_{F_C} = \mu_C$.

## 22. INSIEMI RICORSIVI E RICORSIVAMENTE NUMERABILI

Affrontiamo ora in modo più sistematico la problematica della decidibilità, già discussa per il problema dell'arresto.
La prima nozione che vogliamo introdurre è quella di insieme ricorsivo.
Intuitivamente, un sottoinsieme $A \subseteq N$ è ricorsivo sse esiste un procedimento finito che permette di decidere, per ogni $x \in N$ , se $x \in A$ oppure x non $\in$ A. Formalmente:

*Def 1*: **Funzione caratteristica** $X_A : N \to \{0, 1\}$ di un insieme $A \subseteq N$ è la funzione

$$X_A(x) = \text{se } x \in A \text{ allora } 1 \text{ altrimenti } 0$$

Un insieme A si dice **ricorsivo** sse la sua funzione caratteristica $X_A$ è ricorsiva totale.
Questa definizione si estende facilmente alle relazioni $R \subseteq N * N$.
Una relazione R sarà detta ricorsiva se la sua funzione caratteristica $X_R(x, y)$ è ricorsiva totale.
La nozione di insieme ricorsivo coincide con quella, da noi precedentemente introdotta, di problema decidibile.

Esempio 1. L'insieme dei numeri dispari è ricorsivo. La sua funzione caratteristica è infatti: $y = \langle x \rangle_2$, che è ricorsiva totale.

Esempio 2. Sia *P* un programma RAM. L'insieme di coppie $T_P$ definito da { < x, k > | Il programma P su ingresso x termina dopo k passi} è un esempio di insieme (relazione) ricorsivo.

E' infatti facile costruire un nuovo programma *P* che, accettando come ingresso <x, k>, simula i primi k passi di computazione di *P*. Se dopo k passi *P* si arresta, allora dà come uscita 1, altrimenti 0.

Esempio 3. Sia *P* il programma RAM:

$R_1 \leftarrow \langle R_1, R_1 \rangle$; $R_0 \leftarrow \varphi_u(R_1)$

(da noi precedentemente utilizzato per mostrare la non decidibilità del problema dell'arresto). Sia $A = \{x \mid \exists k(\langle x, k \rangle \in T_P)\}$
Allora A non è ricorsivo. Infatti, esiste k per cui $\langle x, k \rangle \in T_P$ sse il programma *P* si ferma sul dato x, e noi in precedenza avevamo dimostrato che la funzione caratteristica dei valori per cui *P* si arresta non può essere ricorsiva totale.

La seconda nozione che introdurremo è quella di insieme **ricorsivamente numerabile**. Intuitivamente, diremo che un insieme A è ricorsivamente numerabile se esiste un procedimento finito che permette di elencare gli elementi di A "uno dietro l'altro".

*Def.2*. Un insieme $A \subseteq N$ è detto ricorsivamente numerabile se o $A = \phi$, o A è l'immagine di una funzione ricorsiva totale $f : N \rightarrow N$.
L'insieme A, ricorsivamente numerabile può essere pertanto "listato" da:

f(0), f(1), f(2), ..., f(x), ...

Un insieme ricorsivamente numerabile è detto anche parzialmente decidibile perché, per decidere se un elemento $y \in A$, basta enumerare fino al valore x per cui y = f(x). Naturalmente, tale metodo è parziale perché, in generale non è possibile decidere se y non $\in$ A.

Una definizione equivalente di insieme ricorsivamente numerabile è fornita dal seguente:

**TEOREMA**: (1) e (2) sono equivalenti, dove:

(1) A è ricorsivamente numerabile
(2) Esiste una relazione ricorsiva $R \subseteq N * N$ per cui:

$A = \{x \mid \exists k((x, k) \in R)\}$

**Dimostrazione**: (1) -> (2)
Sia A il codominio di *f* : N -> N ricorsiva totale. Posto:
$R = \{(x, k) \mid x = f(k)\}$ vale:
a) R ricorsiva
b) $A = \{x \mid \exists k((x, k) \in R)\}$

(2) -> (1)
Ricordiamo le funzioni coppia $\langle x, y \rangle$ e le due proiezioni sin, des
Sia $\phi \neq A \subseteq N$, e R una relazione ricorsiva per cui:

$A = \{x \mid \exists k(x, k) \in R\}$

Poiché, per ipotesi $A \neq \phi$, esiste un elemento $a \in A$. Definita allora la funzione:

f(n) = se $\langle$ sin n, des n $\rangle \in$ R allora sin n
altrimenti a

Vale:
a) f ricorsiva totale

b) A é immagine di f.

Esempio 2 ci permette allora di concludere che, dato un programma *P*, l'insieme dei valori, per cui la funzione computata da *P* é definita, é ricorsivamente numerabile.

Esempio 3 ci mostra infine un esempio di insieme ricorsivamente numerabile ma non ricorsivo.

Ulteriori caratterizzazioni e proprietà degli insiemi ricorsivamente numerabili sono presentate nel cap.7 di "A Programming Approch to Computability" di Kfoury - Moll - Arbib.

## 23. TEOREMA DI RICE

In questo paragrafo dimostriamo un semplice risultato (il teorema di Rice), che risulta uno strumento molto utile per mostrare che certi problemi di decisione sono algoritmicamente insolubili.

*Def.1*: Dato u sistema di programmazione $\{\varphi_j\}$, un insieme $I \subseteq N$ é detto "rispettare le funzioni" se:

$$i \in I \wedge \varphi_i = \varphi_j \Rightarrow j \in I$$

In altre parole, I rispetta le funzioni se, contenendo un programma i, allora contiene tutti i programmi che calcolano la funzione $\varphi_i$.
Il teorema di Rice asserisce che, salvo casi triviali, un insieme I che rispetta le funzioni non può essere ricorsivo:

**TEOREMA (di Rice)**: Supponiamo che $I \subseteq N$ rispetti le funzioni. Allora I é ricorsivo se e solo se $I = \varphi$ o $I = N$.

Dimostrazione: Sia $\acute{\imath} \neq I \neq N$. Supponiamo che I sia ricorsivo, e mostriamo che I non preserva le funzioni.
Poichè, $\acute{\imath} \neq I \neq N$, riesco a trovare $i \in I$ e $c$ non $\in I$.

Consideriamo la funzione:
$t(x) =$ se $x \in I$ allora c
altrimenti i
Poichè, abbiamo supposto che I sia ricorsivo, t(x) risulta una funzione ricorsiva totale. Per il teorema di ricorsione, esiste allora un indice j per cui: $\varphi_j = \varphi_{t(j)}$.
Ma, per come é costruita t, vale che $j \in I$ sse t(j) non $\in I$.
Ciò significa che I non preserva le funzioni.

L'interesse del teorema di Rice é legato alla seguente osservazione:

Sia data una proprietà delle funzioni computabili (per esempio: f é totale, oppure f é iniettiva, ecc.) Sia I l'insieme dei programmi che calcolano funzioni con quella data proprietà.
Allora il teorema di Rice stabilisce il risultato negativo che, salvo i casi triviali I = $\varphi$ o I = N, I non é ricorsivo.
In altre parole, non é algoritmicamente possibile dedurre in generale, dato un programma, se la funzione computata verifica o meno una proprietà preassegnata!