

1^a Lezione

Sapere se un problema ammette soluzione algoritmica o meno non è facile. Infatti, vi sono molti problemi che, come vedremo, non sono risolubili da un algoritmo. Ad esempio, non esiste un programma che riesca a dire se un programma termina sempre o no, inoltre non esiste un programma che riesca a dire se un programma fa sempre quello che deve fare (**problema della correttezza**). L'obiettivo che quindi ci poniamo è, per ora, dimostrare che esistono problemi non risolubili con un algoritmo.

1.1 Un po' di terminologia

Se abbiamo un insieme $A = \{a_1, \dots, a_k\}$ finito che chiamiamo **alfabeto**, possiamo costruire l'insieme di tutte le stringhe generabili a partire da A. Chiameremo tale insieme infinito A^* . Ad esempio se $A = \{a, b\}$, allora le parole

a
aa
aaa
aaaa
aaaaa
.....

che come si può intuire è un insieme infinito.
D'ora in poi supponiamo di lavorare con i numeri naturali come alfabeto.

1.2 Come può essere visto un programma

Un programma, in generale, può essere visto come un qualcosa che calcola una funzione $f : N \rightarrow N$, dove quindi l'input e l'output del programma sono considerati sottoinsiemi di N. Vedremo che ci sono funzioni $f : N \rightarrow N$ che non sono calcolate da alcun programma (i programmi sono di meno). Ricordiamo che trovare una soluzione algoritmica significa trovare un programma che possa implementarla.

Se un problema non ammette soluzione algoritmica significa che tutti i programmi scrivibili non implementano la soluzione algoritmica, il che è più difficile da dimostrare perché bisognerebbe provare tutti i programmi uno a uno. Ad esempio, il problema di calcolare $y = x^2$ ammette soluzione algoritmica, basta implementare il programma adatto, mentre dimostreremo che invece ci sono altri problemi non risolubili, come quelli accennati al punto 1.1.

Abbiamo visto che i programmi, in fin dei conti, sono stringhe, per cui se dimostro che posso enumerare le stringhe allora, tanto più, potrò enumerare i programmi sintatticamente corretti, che sono un loro sottoinsieme. Le stringhe derivanti da un alfabeto $A = \{a, b\}$ le posso enumerare così:

$0 \rightarrow \lambda$
 $1 \rightarrow a$
 $2 \rightarrow b$
 $3 \rightarrow aa$
 $4 \rightarrow ab$
 $5 \rightarrow bb \dots$ ecc...

In questo modo, cosi', possiamo enumerare I programmi, che sono un sottoinsieme delle stringhe. Quindi I problemi che ammettono soluzione algoritmica, cioe' I programmi, sono enumerabili.

Scopriremo, pero', che l'ordine di grandezza del numero dei problemi e' molto maggiore di quello dei problemi risolubili, cioe' dei programmi, per cui ci sono molti piu' problemi insolubili rispetto a quelli solubili.

1.3 Problemi risolubili e non risolubili

Il nostro obiettivo sara' quindi dimostrare che $F = \{f : N \rightarrow N\}$ e' piu' grande di $F_r = \{f : N \rightarrow N \text{ tale che } f \text{ e' risolubile}\}$. A questo scopo, innanzitutto vediamo alcuni esempi che ci consentono di correlare le funzioni $f : N \rightarrow N$ ai numeri reali dell'intervallo $[0,1]$, e cioe':

- $f(x) = x^2$.

In questo caso, tale funzione puo' essere rappresentata dalla sequenza $0,1,4,9,16,25,36\dots$. Ad esempio, per $x=5$ bastera' considerare il sesto elemento della sequenza (che parte da indice 0), ovvero 25. Ricordiamo che le funzioni che qui stiamo considerando sono di tipo intero.

- $f : N \rightarrow \{0,1,2,3,4,5,6,7,8,9\}$, tale che, ad esempio, $f(1) = 1$, $f(8) = 8$, $f(10) = 0$, $f(18) = 8$.

Come nell'esempio precedente, anche in questo caso la funzione potra' essere rappresentata da una sequenza, che in questo caso sara' $0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,\dots$

Quindi, concludiamo che qualsiasi funzione del tipo $f : N \rightarrow \{0,\dots,9\}$ puo' essere rappresentata da infinite ripetizioni del codominio (non necessariamente nello stesso ordine).

Un numero compreso tra 0 e 1 si puo' quindi ottenere da una funzione $f : N \rightarrow \{0,\dots,9\}$ aggiungendo uno zero all'inizio della sequenza e togliendo le virgole. Ad esempio, dal secondo esempio, otterro' il numero $0,1234567890123\dots$, che e' appunto un numero reale compreso tra 0 e 1. Ovviamente e' vero anche il contrario, ad esempio da $0,512340123$ otteniamo la sequenza $5,1,2,3,4,0,1,2,3$.

Questo ci fa capire che le funzioni $f : N \rightarrow \{0,\dots,9\}$ sono in corrispondenza con I numeri reali tra 0 e 1, perche' ciascun numero reale puo' essere rappresentato da una sequenza di numeri da 0 a 9, e quindi dalla funzione che genera tale sequenza. Detto questo, sappiamo che l'insieme dei reali tra 0 e 1 non e' numerabile (e' piu' grande di N , infatti tra due numeri reali di un ipotetico elenco, posso sempre trovarne uno nuovo), quindi anche l'insieme delle funzioni $f : N \rightarrow \{0,\dots,9\}$ ha cardinalita' maggiore di N . Da cio', per estensione, deduciamo che anche l'insieme delle $f : N \rightarrow N$ ha cardinalita' maggiore di N .

Nel paragrafo 1.2 avevamo visto che un problema puo' essere visto sotto forma di funzione $f : N \rightarrow N$, mentre nell'1.1 avevamo visto che I problemi che ammettono soluzione algoritmica (programmi) sono numerabili, sono cioe' in corrispondenza con N . Da cio' deduciamo che I problemi che ammettono soluzione algoritmica sono molti meno della somma totale dei problemi, *per cui ci saranno molti problemi (la maggior parte) che non ammetteranno soluzione algoritmica*.

1.4 Formalismi e paradigmi

Diremo che una funzione e' calcolabile se essa e' risolubile con un algoritmo realizzato con un linguaggio (formalismo) qualsiasi. Da cio' segue che *tutti I formalismi sono equivalenti, cioe' hanno la stessa potenza risolutiva*.

Vedremo di seguito due paradigmi di programmazione: l'uno imperativo (macchine di Turing ed a registri) e l'altro funzionale. Entrambi, come abbiamo visto, hanno la stessa potenza risolutiva perche' il concetto di funzione calcolabile e' assoluto.

1.5 Automi

Un **automa generale** e' composto da:

Dati	L'insieme di dati in input
Risultati	L'insieme dei risultati in output
Ingresso	La funzione ingresso
Uscita	La funzione uscita
Configurazioni	L'insieme di configurazioni $\{c_0, \dots, c_n\}$
Trasformazione	La funzione di trasformazione

} in genere infinito
→ numeri di trasformazione periale

Vediamo quindi quali funzioni agiscono su un certo dato ($d \in \text{Dati}$) quando viene dato in ingresso all'automa:

- **Ingresso:** trasforma I dati in ingresso nella configurazione iniziale, che e' il tipo di dato accettato dalla macchina. Formalmente Ingresso (d) = c_0 , dove c_0 e' la configurazione iniziale.
- **Trasformazione:** trasforma configurazioni in configurazioni. Formalmente avremo $c_1 = \text{trasf}(c_0)$, $c_3 = \text{trasf}(c_2)$, $c_n = \text{trasf}(c_{n-1})$, ecc.
- **Uscita:** trasforma la configurazione finale nel risultato che poi sara' fornito in output, cioe' , formalmente, risultato = Uscita(c_n).

L'esecuzione del programma, dunque, avviene grazie a una sequenza di configurazioni c_0, \dots, c_n attraverso le quali si passa tramite la funzione di trasformazione. *Tale funzione non e' totale, bensì parziale*, in quanto esiste una configurazione (c_n) che termina la computazione e a cui la funzione di trasferimento non puo' essere applicata. In proposito ricordiamo che una funzione $f : A \rightarrow B$:

- **e' strettamente parziale se** $\{x \in A \mid f(x) \in B\} \subsetneq A$
- **e' totale se** $\{x \in A \mid f(x) \in B\} = A$

trasformazione

1.6 Automi a programma

Diremo che un **automa a programma** e' un automa generale in cui la funzione di trasformazione e' determinata da un certo programma P .

In questo senso, il programma P prende una configurazione c_i e la traduce in una istruzione da dare in pasto alla macchina: esso dice quindi che istruzione applicare alla configurazione per trasformarla nella configurazione successiva. Dunque un automa a programma e' costituito da (Dati, Risultati, Ingresso, Uscita, Configurazioni, P , Istruzioni elementari), con P che puo' essere visto come una funzione cosi' fatta:

$$P : \text{Conf} \rightarrow \text{Istruzioni elementari} \quad (\text{funzione, in genere, periodica})$$

Il funzionamento, che nel caso della funzione di trasformazione consisteva semplicemente nell'applicare tale funzione alle configurazioni, col programma si munisce di un ulteriore passo, ovvero il determinare le funzioni elementari. In definitiva, quindi, ci troveremo di fronte al seguente processo:

$$P(c_0) = e_0 \text{ (istruzione elementare)}$$

$$c_1 = e_0(c_0)$$

$$P(c_1) = e_1$$

$$c_2 = e_1(c_1) \quad \text{(istruzione elementare)}$$

*ei è una funzione totale che
configurazioni in configurazioni*

dove e_i e' alla fine una funzione che trasforma configurazioni in configurazioni.

Diremo che la computazione termina quando $P(c)$ e' **indefinito** ($P(c) \uparrow$), mentre continua se e' **definito** ($P(c) \downarrow$). Dunque per un automa il ciclo di funzionamento e' il seguente:

```

FUNCTION Automa (d:Dati):Risultati;
VAR c:configurazione;
BEGIN
  c := Ingresso(d);
  WHILE (P↓c) DO
    c := (P(c))(c); (dove P(c)=e, istruzione elementare)
    Automa := Uscita(c);
  END;

```

2^a Lezione

Un automa, come detto, si comporta secondo un modello di calcolo imperativo. Cio' vuoldire che in tale modello gli accessi alla memoria sono diretti. Vediamo ora un primo esempio di modello di calcolo imperativo, ovvero la macchina di Turing.

2.1 Macchina di Turing

Supponiamo di voler calcolare la somma tra due numeri naturali, ad esempio $57 + 48$. Quello che, naturalmente, ci verrebbe spontaneo fare, dovendo enunciare un algoritmo elementare per effettuare tale operazione, sarebbe mettere in colonna I 2 numeri, posizionarsi sul 7 di 57, leggere tale cifra, ricordarsela, spostarsi sull'8 di 48, leggere tale cifra, spostarsi ulteriormente sotto e, una volta visto che le cifre da sommare sono terminate, scrivere il resto di $15/10$ e riportare un 1 per le operazioni successive.

Questo semplice algoritmo puo' essere implementato anche tramite una macchina di Turing, la quale consiste di una memoria infinita (in questo caso bidimensionale), e di una testina che puo' compiere solamente tre operazioni, e cioe' leggere il contenuto di una cella di memoria, scrivere in una cella di memoria, spostarsi ad una cella di memoria contigua (secondo I punti cardinali).

Si puo' dimostrare che una macchina di questo tipo, se opportunamente programmata, puo' calcolare tutto cio' che e' calcolabile, quindi puo' implementare qualsiasi algoritmo.

Esempio 2.1

Consideriamo il caso appena visto dell'operazione $57 + 48$. Inizialmente la macchina si trovera' in una situazione in cui non e' ancora stato letto nulla. In questo caso la testina viene posta sul 7 di 57, in uno stato, che chiameremo q_0 , in cui non e' necessario tener conto di alcun riporto dovuto ad operazioni precedenti. Successivamente, nel caso al termine della somma delle cifre di una colonna si dovesse riportare 1, si passera' allo stato q_1 , nel caso si dovesse riportare 5 allo stato q_5 e cosi' via. Inoltre, ogni volta che ci si sposta lungo una colonna, bisognera' aver memoria delle cifre lette fino ad allora lungo quella colonna, per cui, quando la testina si sposterà sull'8 di 48, la macchina si sposterà in uno stato $q_{0,7}$, nel senso che ancora non ci sono riporti da considerare, ma gia' vi e' un 7 di cui dover tener conto quando si arriverà al termine della colonna. Vediamo praticamente come si cambia stato osservando la figura sottostante:

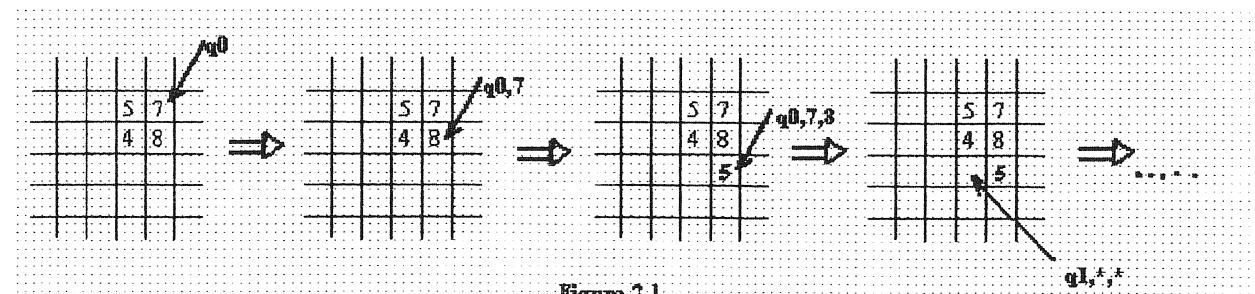


Figura 2.1

Nella figura 2.1, si nota che, letto il primo carattere blank, si scrive il resto della somma $7+8$ e ci si sposta a sinistra, portandosi nello stato $q_{1,*,*}$. Qui gli asterischi stanno ad indicare che, successivamente, dovrò muovermi "a vuoto" di due celle verso l'alto, in modo da ritornare in cima alla nuova colonna e poter così ricominciare con le operazioni fatte in precedenza per la prima colonna.

Vediamo quindi, nella tabella sottostante, il giusto ordine in cui vengono eseguite le operazioni dalla macchina:

Configurazione da cui parto		Azioni che commetto e stato a cui vado		
Stato	Cosa leggo	Cosa scrivo	Nuovo stato	Direzione testina
q_0	c (cifra)	c	$q_{0,c}$	S (Sud)
$q_{0,c}$	d (cifra)	d	$q_{0,c,d}$	S
$q_{0,c,d}$	Blank	(c+d) mod (10)	q_1^{**}	O

In pratica questa tabella costituisce la prima parte del programma che esegue la somma di due numeri naturali di due cifre tramite macchina di Touring. E' importante notare che questo programma potrebbe essere stato scritto in molti modi diversi, e molte delle notazioni che abbiamo usato sono puramente arbitrarie (la lettera q, gli asterischi per indicare di andare su, ecc).

Ora che abbiamo visto con un esempio in cosa consiste una macchina di Touring, vediamo di formalizzare tutto quanto e' stato detto fino ad ora.

2.2 Descrizione formale di una macchina di Touring

Una macchina di Touring e' un oggetto del tipo $M = (Q, q_0, A, P)$, dove:

- Q e' l'insieme degli stati cui puo' passare la macchina
- q_0 e' lo stato iniziale ($q_0 \in Q$)
- A e' l'alfabeto dei caratteri che posso leggere o scrivere (blank $\in A$, come abbiamo visto e' utile)
- P e' il programma, che e' un insieme di quintuple del tipo $(q, a, b, q', direzione)$ dove:
 - q e' lo stato da cui si parte
 - a e' cio' che leggo con la testina
 - b e' cio' che scrivo nella cella prima di spostarmi (il dato stesso se devo lasciarlo invariato)
 - q' e' lo stato cui passo appena prima di spostarmi
 - $direzione$ e' la direzione in cui si muove la testina alla fine dell'istruzione. Questa sara' destra, sinistra nel caso di una memoria monodimensionale, I punti cardinali nel caso di una memoria bidimensionale.

Quindi possiamo concludere che la macchina di Touring e' una macchina a basso livello senza accessi diretti alla memoria, ma che viene considerata ugualmente operante secondo il paradigma imperativo, in quanto vi e' un accesso diretto alla cella di memoria sotto la testina.

Esempio 2.2

Supponiamo di avere un nastro unidimensionale e di lavorare con cifre binarie. Cerchiamo un programma per cancellare le prime due occorrenze del carattere 1:

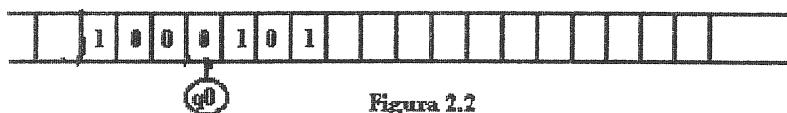


Figura 2.2

L'unico problema si presentera' quando nell'output non troveremo 2 uni. In questo caso bisognera' scegliere una strategia opportuna, in modo da non inoltrarci in un loop infinito. Ad esempio ci si potrebbe fermare dopo un po' di zeri (si consideri che, se il blank non e' contemplato negli stati, qualora venisse incontrato la macchina si fermerebbe).

Vediamo quindi il programma che implementa l'algoritmo:

Stato iniziale	Cosa leggo	Cosa scrivo	Stato finale	Direzione
q_0	0	0	q_0	D
q_0	1	Blank	q_1	D
q_1	0	0	q_1	D
q_1	1	Blank	q_2	D

Quindi, la macchina si ferma quando raggiunge lo stato q_2 , in quanto nessuna istruzione del programma contempla una situazione in cui esso sia lo stato di partenza.

Un'altra importante considerazione da fare e' che *un programma per una macchina di Touring non ha una sequenzialita' nell'eseguire le istruzioni, bensì e' costituito da tante istruzioni (paragonabili a degli IF-THEN) di uguale importanza ed ordine ininfluente*.

Vediamo ora come la macchina di Touring puo' essere definita un automa a programma.

2.3 La macchina di Turing come automa a programma

Vediamo di ricavare le caratteristiche dell'automa a programma nel caso della macchina di Turing. In questo caso le istruzioni elementari sono le quintuple, mentre il tipo di dato del tipo Configurazione sara' costituito, nel caso di macchina monodimensionale, dal nastro, lo stato della macchina e la posizione della testina.

Un modo per rappresentare una configurazione della macchina di Turing di figura 2.2 potrebbe essere quello di scrivere la sequenza $100q_00101$, che rappresenta la memoria nel momento in cui la testina si trova sullo 0 posto subito dopo q_0 , che e' lo stato in cui si trova momentaneamente la macchina.

Il metodo ufficiale con cui pero' si rappresenta una configurazione della macchina di Turing e' la rappresentazione tramite la tripla (memoria, stato, posiz.testina), nella quale, come si vede in figura 2.3, le celle di memoria sono numerate:

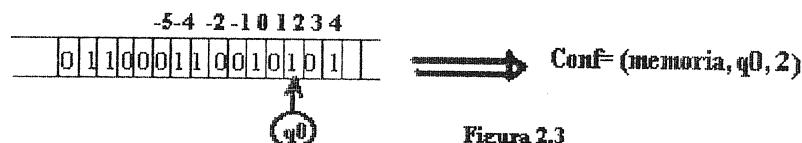


Figura 2.3

Se la posizione della testina e' posta oltre l'area del nastro in cui sono scritti dati, l'informazione sara' blank, ad esempio, se chiamiamo m la memoria, potremo scrivere $m(2) = 1$ ed $m(100000) = \text{blank}$, quindi la memoria puo' essere vista come una funzione.

Un'altra considerazione utile e' che, essendo la memoria infinita, anche l'insieme di configurazioni sara' infinito. L'unica cosa sicuramente finita sara' il numero di stati, e questo perche' il programma e' finito.

Se ho un'istruzione elementare $\{q, a, b, q', D\}$ e una configurazione (m, q'', i) a cui la applico, otterro' la configurazione:

$$\{q, a, b, q', D\} (m, q'', i) = \begin{cases} (m[b/i], q', i+1) & \text{se } (q'' = q) \text{ AND } (m(i) = a) \\ (m, q'', i) & \text{altrimenti} \end{cases}$$

Dove $m[b/i]$ significa che nella cella i -esima scrivero' b .

Se applichiamo invece P ad una configurazione, secondo quanto visto nel paragrafo 1.6 dovremmo generare una istruzione elementare, la quale dovrà poi essere applicata alla stessa configurazione per ottenere la successiva. Infatti, compiendo tale operazione otteniamo:

$$P(m, q, i) = \begin{cases} \{q, m(i), b, q', D\} & \\ \uparrow & \text{se in } P \text{ non esiste una quintupla che cominci per } q, m(i) \end{cases}$$

Nel caso dell'esempio 2.2 se $q=q2$ (stato finale) non vi saranno istruzioni che contemplino tale possibilita' per cui il programma termina.

3^a Lezione

Di seguito vedremo una nuova macchina ad accesso diretto, ovvero la macchina a registri, mentre successivamente ci soffermeremo anche su un modello di calcolo di tipo funzionale.

3.1 La macchina a registri

Anche la macchina a registri, come la macchina di Turing, e' ad accesso diretto. In questo caso, pero', nel vero senso della parola, in quanto si hanno a disposizione n registri (x_1, \dots, x_n) cui si puo' accedere direttamente. Ciascuno di questi registri puo' contenere un numero naturale, quindi potenzialmente e' infinito.

Un **programma** per questa macchina sara' composto da una serie di istruzioni:

I_1
 I_2
—
 I_k

Tali istruzioni faranno parte di uno dei seguenti 4 tipi:

- **Assegnazione:** ($x := 0$)
- **Somma:** ($x := x + 1$)
- **Trasferimento:** ($x_i := x_j$)
- **Salto Condizionato:** (if $x_i = 0$ then goto q)

Si puo' dimostrare che con queste istruzioni e' possibile implementare qualsiasi algoritmo.

Con questa macchina, un programma puo' essere visto come una funzione del tipo $N^k \rightarrow N^r$, il che indica che i dati in input vengono messi nei primi k registri, mentre i dati in output possono essere letti nei primi r registri.

3.2 La macchina a registri come automa a programma

Prima di tutto riprendiamo il concetto di configurazione, adattandolo alla macchina a registri. Una configurazione di questa macchina consiste, essenzialmente, nel contenuto della memoria e del numero di istruzione che il programma sta eseguendo (**contatore di programma**). La memoria, essendo composta da n registri, sara' una sequenza di naturali lunga n. Quindi, se per ipotesi avessimo tre registri che contenessero i numeri 10, 20, 30, e il programma stesse eseguendo l'istruzione 5, potro' rappresentare la configurazione come la sequenza (5, 10, 20, 30).

Da quanto si e' visto, quindi, forse la piu' grande differenza tra questa macchina e quella di Turing e' che qui le istruzioni vengono eseguite secondo un certo ordine (e il contatore di programma tiene conto in questo) mentre nella macchina di Turing ogni istruzione poteva teoricamente essere eseguita in qualsiasi momento.

Vediamo, ora, come i quattro tipi di istruzioni di una macchina a registri si comportano una volta applicati ad una configurazione. Come convenzione, rappresenteremo una configurazione con la sequenza (m_0, m_1, \dots, m_n) , dove m_0 sara' il contenuto del contatore di programma, mentre il resto della sequenza consistera' nel contenuto dei singoli registri di memoria nell'ordine rispettivo.

Figura 3.1

$$1) I_i \equiv x_j := 0 \text{ allora } || I_i || (m_0, m_1, \dots, m_n) = \begin{cases} (m_0+1, m_1, \dots, m_{j-1}, 0, m_{j+1}, \dots) & \text{per } (m_0 = i) \\ (m_0, m_1, \dots, m_n) & \text{altrimenti} \end{cases}$$

$$2) I_i \equiv x_j := x_{j+1} \text{ allora } || I_i || (m_0, m_1, \dots, m_n) = \begin{cases} (m_0+1, m_1, \dots, m_{j-1}, m_{j+1}, \dots) & \text{per } (m_0 = i) \\ (m_0, m_1, \dots, m_n) & \text{altrimenti} \end{cases}$$

$$3) I_i \equiv x_j := x_r \text{ allora } || I_i || (m_0, m_1, \dots, m_n) = \begin{cases} (m_0+1, m_1, \dots, m_{j-1}, m_r, \dots) & \text{per } (m_0 = i) \\ (m_0, m_1, \dots, m_n) & \text{altrimenti} \end{cases}$$

$$4) I_i \equiv \text{if } x_j = q \text{ allora } || I_i || (m_0, m_1, \dots, m_n) = \begin{cases} (q, m_1, \dots, m_n) & \text{per } (m_0 = i) \text{ e } (m_j = q) \\ (m_0+1, m_1, \dots, m_n) & \text{per } (m_0 = i) \text{ e } (m_j \neq q) \\ (m_0, m_1, \dots, m_n) & \text{altrimenti} \end{cases}$$

Infine, applichiamo P alla configurazione, con $P(m_0, m_1, \dots, m_n) = \begin{cases} || I_m || & \text{per } m_0 < k \\ \uparrow & \text{altrimenti} \end{cases}$

Dalla figura 3.2 possiamo capire sia come ciascuna istruzione modifichi le configurazioni, sia come il programma, applicato alla configurazione corrente, determini una istruzione elementare. Il ciclo di funzionamento e' ovviamente lo stesso di qualsiasi altro automa a programma.

Si puo' dimostrare che e' possibile simulare una macchina di Turing con una a registri e viceversa, ottenendo programmi equivalenti. Cio' puo' essere fatto compilando I programmi di una macchina con un'altra macchina e viceversa.

Teniamo a precisare che il nostro scopo non sara' di scrivere programmi con tali macchine, bensì dimostrare che si possano scrivere, ovvero dimostrare che vi e' o non vi e' una soluzione algoritmica per certi problemi.

3.3 Un modello di calcolo funzionale

A questo punto, cambiamo stile di programmazione, e cioe' passiamo da uno stile imperativo ad uno funzionale, pur tenendo presente che I linguaggi funzionali, essendo piu' lontani dal linguaggio macchina, avranno compilatori piu' complicati e quindi meno efficienti..

Coi linguaggi imperativi accediamo alla memoria direttamente (tramite variabili) come ad esempio nelle macchine a registri, mentre con quelli funzionali considereremo strutture in cui la memoria non e' suddivisa in parti accessibili, bensì e' vista come un unico blocco che viene modificato da ogni singola istruzione del programma. Da questo deduciamo che avremo una memoria iniziale, delle istruzioni che la modificano ed una memoria finale.

Dal momento che lavoriamo con I naturali, la memoria sara' costituita da una sequenza di naturali, che quindi verrà trasformata ad ogni istruzione.

Da queste premesse formiamo il nostro linguaggio funzionale, che avra' le seguenti funzioni base:

- **Successore:** $S : N \rightarrow N$ con $S(x) = x + 1$
- **Costrettore dello zero:** $0 : N^0 \rightarrow N$ con $N^0 = \{\lambda\}$, ovvero il simbolo nullo, per cui si parte da nessun input, per cui viene trasformata una sequenza lunga zero in una sequenza lunga uno, costituita dallo zero.
- **Funzione di proiezione:** $P_{i,j}^n : N^n \rightarrow N^{(j+1)}$. Questa funzione in pratica prende I pezzi della memoria che non sono tra gli indici i,j e li scarta. Ad esempio:

$$5,73,21 \xrightarrow{P_{1,2}^3} 5,73$$

oppure

$$3,1,2,45,5 \xrightarrow{P_{3,3}^5} 2$$

In generale, la proiezione non avra' senso qualora $i > j$. In questo caso l'output sara' λ .

3.4 Funzioni piu' avanzate nel nostro paradigma funzionale.

Vedremo adesso delle funzioni che renderanno decisamente piu' potente, anche se non potentissimo, il nostro modello funzionale.

Partiamo dalla funzione che permette di concatenare piu' funzioni (che rappresenteremo con ;). In pratica e' una vera e propria **composizione di funzioni**, la quale ovviamente si potra' applicare solo se il dominio della seconda funzione corrispondera' al codominio della prima. Formalmente, dati $f : N^n \rightarrow N^k$ e $g : N^k \rightarrow N^r$ avremo:

$$(f;g)(x_1, \dots, x_n) = g(f(x_1, \dots, x_n)) \quad \text{Quindi un programma potrebbe essere } S;S(x) = S(S(x)) = S(x+1) = x+2$$

Per costruire un 2, quindi, eseguiro' la funzione composta $\emptyset;S;S$

Vediamo ora un costrutto che permette di lavorare in parallelo, e che indicheremo con \parallel (**parallelo**). Ovvero, avendo due programmi che lavorino su memorie diverse, e' possibile unificare il risultato. In questo modo e' possibile incrementare la grandezza di una memoria, mentre con la proiezione abbiamo visto che la restringevamo. Ad esempio, se ho $f : N^n \rightarrow N^k$ e $g : N^m \rightarrow N^r$ allora avremo:

$$(f\parallel g) : N^{n+m} \rightarrow N^{k+r} \text{ tale che } (f\parallel g)(x_1, \dots, x_n, y_1, \dots, y_m) = f(x_1, \dots, x_n), g(y_1, \dots, y_m)$$

Ad esempio, se eseguo $(\emptyset;S;S)\parallel(\emptyset;S;S;S) = 2,3$, che e' una memoria lunga 2, quindi sono riuscito ad ampliarla.

Un altro costrutto utile e' la **giustapposizione** (indicata con \wedge), la quale si presenta molto simile allo schema funzionale parallelo, solo che in questo caso i due calcoli paralleli lavorano sullo stesso input. E' in pratica equivalente a un calcolo parallelo con input duplicato. Formalmente, con $f : N^n \rightarrow N^k$ e $g : N^n \rightarrow N^r$ avremo:

$$(f^\wedge g) : N^n \rightarrow N^{k+r} \text{ tale che } (f^\wedge g)(x_1, \dots, x_n) = f(x_1, \dots, x_n), g(x_1, \dots, x_n)$$

e' chiaro che anche con questo costrutto viene allargata la memoria.

Introduciamo infine il costrutto **exp**/il quale, in pratica, rappresenta un costrutto di tipo FOR. Data $f : N^e \rightarrow N^n$, otteniamo:

$$\exp(f) : N^{n+1} \rightarrow N^n, \text{ dove } \exp(f)(x_1, \dots, x_n, y) = f^y(x_1, \dots, x_n) \quad \text{e con } f^0(x_1, \dots, x_n) = x_1, \dots, x_n$$

$$e \quad f^{y+1}(x_1, \dots, x_n) = f(f^y(x_1, \dots, x_n))$$

In pratica, in questo costrutto, y rappresenta il numero di volte che deve essere eseguito il ciclo FOR. Facciamo un esempio di cosa si puo' fare con un costrutto del genere:

Prendiamo $S;\exp$, che equivale ad $\exp(S)(x) : N^2 \rightarrow N$, cioe' $x,y \rightarrow S^y(x) = x+y$, per cui abbiamo in pratica costruito l'operatore per eseguire la somma.

A questo punto, avendo il costrutto FOR, siamo in grado di scrivere programmi piu' complessi, ma non ancora potentissimi, in quanto ci manca ad esempio un qualcosa che assomigli ad un costrutto WHILE-DO (ciclo condizionato).

4^a Lezione

Abbiamo visto come sono definiti alcuni costrutti funzionali fondamentali. Vediamo adesso qualche esempio che ci sia di aiuto nel capire come essi possano essere adoperati.

4.1 Esempi di programmi che utilizzano il formalismo funzionale

Vediamo tre esempi che adottano il formalismo funzionale. Nei primi due saranno maggiormente esposti i costrutti di proiezione, giustapposizione e calcolo parallelo, nel terzo si porra' maggiormente l'attenzione sul costrutto $\exp(f)$.

4.1.1 Come utilizzare parti di memoria

Si abbia la memoria $\text{mem} = (5, 3, 7, 8, 10, 12)$. Il problema che ci poniamo e' sommare il terzo ed il quarto elemento di questa, ottenendo $\text{mem2} = (5, 3, 15, 10, 12)$.

Quello che ci viene piu' spontaneo fare, in questo caso, e' copiare il 5 il 3, effettuare la somma $7+8=15$ e copiarla nella terza cella di memoria, infine copiare il 10 e il 12, ottenendo cosi' quello che ci eravamo prefissati. Per implementare questo algoritmo, ci accorgiamo che e' sufficiente suddividere le 3 operazioni di cui sopra in 3 calcoli paralleli, che si possono riassumere nel seguente programma:

$$P: P^2_{1,2} \parallel + \parallel P^2_{1,2}$$

dove la prima proiezione copia il 5 e il 3, la seconda somma compie l'operazione $(7+8)$ e concatena il risultato al precedente (si noti che in questo caso l'operatore somma prendera' in considerazione le due celle che vogliamo senza dover compiere prima alcuna proiezione, questo perche' operiamo in modo parallelo), ed infine l'ultima proiezione copia il 10 e il 12. Poiche' $P^2_{1,2}: N^2 \rightarrow N^2$ e $+: N^2 \rightarrow N$, allora il nostro programma $P: N^6 \rightarrow N^5$, come ci aspettavamo. Quindi, in pratica, ogni volta che lo applichiamo, P suddivide la memoria in 3 parti e a queste applica i tre sottoprogrammi $P^2_{1,2}; +; P^2_{1,2}$ separatamente.

4.1.2 Risolvere il problema precedente usando lo schema di giustapposizione

La differenza sostanziale che si riscontra nel risolvere il problema precedente con lo schema di sovrapposizione, e' che in questo caso l'input dei sottoprogrammi deve essere lo stesso. La soluzione quindi sara':

$$P: P^6_{1,2} \wedge (P^6_{3,4}; +) \wedge P^6_{5,6}$$

dove la prima proiezione copia il 5 e il 3, mentre il secondo membro si occupa di prelevare il 7 e l'8 dalla sequenza e sommarli (si noti che prima di utilizzare la somma e' necessario utilizzare una proiezione per prelevare i dati che ci interessano, in quanto la giustapposizione utilizza in entrata tutto l'input a disposizione). L'ultima proiezione ha il compito di copiare il 10 e il 12. Ricordiamo a proposito che le proiezioni, in questo caso, sono del tipo $N^6 \rightarrow N^2$. Ragionando cosi' si puo' vedere come in molti casi, con il paradigma funzionale, si riesca a vedere se una funzione e' calcolabile anche senza scrivere il programma relativo.

4.1.3 Programma per il calcolo del prodotto

Per quanto riguarda il calcolo del prodotto, cio' che vogliamo e' creare una funzione $*: \mathbb{N}^2 \rightarrow \mathbb{N}$, che prenda in ingresso due naturali x, y e ne restituisca il prodotto.

Il problema che ci siamo posti appare avere una soluzione abbastanza immediata, ossia applicare l'esponenziazione alla somma, di modo che attraverso un ciclo di somme si arrivi ad un prodotto. Purtroppo, tale soluzione e', in questi termini, impraticabile, in quanto la somma riduce la memoria da due elementi ad uno, mentre $\exp(f)$ puo' essere applicata solo a funzioni che non modificano le dimensioni della memoria, quindi nell'applicazione ci troveremo di fronte ad un errore sintattico.

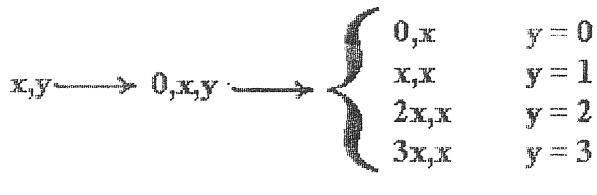
Una soluzione per risolvere questo problema e' di trasformare leggermente la funzione che compie la somma. In particolare, la funzione che itereremo sara' del tipo

$$f : \mathbb{N}^2 \rightarrow \mathbb{N}^2 \text{ con } f =_{\text{def}} (+ \wedge P^2_{2,2}) \\ f(\text{somma}, x) = (\text{somma} + x, x)$$

dove in pratica somma e' una variabile che viene inizializzata a zero e che a programma terminato contiene il risultato del prodotto, mentre x e' una costante che contiene il valore di x (nel prodotto x^*y) e che viene sommata alla variabile somma ad ogni iterazione, oltre che venire riscritta sempre come seconda cella di memoria. Quindi, in pratica, somma si preoccupa di cumulare le somme, mentre x ricorda che bisogna sommare x .

Quindi il ciclo del programma sara' del tipo:

Figura 4.1



Quindi, in definitiva, il programma sara': $*(x,y) =_{\text{def}} (0 \parallel P^2_{1,2}) ; \exp(+ \wedge P^2_{2,2}) ; P^1_{1,1}$

Una esecuzione del programma apparira' come si puo' vedere qui sotto:

$*(2,2) \rightarrow 0,2,2 \rightarrow$	$\rightarrow 4$
$f : 0,2,2 \rightarrow 2,2 \rightarrow 4,2$	

4.2 Altri semplici costrutti funzionali

Passiamo ora a vedere un paio di nuovi costrutti funzionali. Il primo ci permette di **duplicare una cella di memoria**, viene indicato con **D**, ed e' implementato dal seguente programma:

$$D(x) = [(0 \parallel 0 \parallel P^1_{1,1}) ; \exp(S \parallel S)]$$

In pratica viene presa la cella di memoria da duplicare come indice di un ciclo FOR, dentro al quale vengono create, per incrementi successivi, due celle che alla fine conterranno il valore di x . In pratica l'andamento della funzione sara' del tipo $x \rightarrow 0,0,x \rightarrow x,x$.

Un altro semplice costrutto e' quello che permette di calcolare la potenza di un numero. Viene indicato con **Pot** e, come si puo' facilmente intuire, la sua implementazione sara' analogica a quella del prodotto, soltanto che in questo caso dovremo inizializzare la cella risultato a 1, e non a zero. Questo algoritmo puo' essere implementato da:

$$\text{Pot}(x,y) =_{\text{def}} ((0;S) \parallel P^2_{1,2}) ; \exp(* \wedge P^2_{2,2}) ; P^1_{1,1}$$

Anche qui, come nel caso del prodotto, abbiamo una prima cella di memoria che serve per tenere il risultato, e la seconda che ci ricorda il valore di x da moltiplicare ogni volta. Una esecuzione del programma potra' essere:

$\text{Pot}(2,3) \rightarrow 1,2,3 \rightarrow$	$\rightarrow 8$
$f : 1,2,3 \rightarrow 2,2 \rightarrow 4,2 \rightarrow 8,2$	

4.3 Schema funzionale ricorsivo primitivo

In generale, diremo che una funzione f si puo' definire tramite **ricorsione primitiva** se esistono due funzioni:
 $g: \mathbb{N}^r \rightarrow \mathbb{N}^r$
 $h: \mathbb{N}^{r+\alpha} \rightarrow \mathbb{N}^r$ tali che

$$f = \text{rec } (g, h) = \begin{cases} f(\underline{x}, 0) = g(\underline{x}) & \text{Base dell'induzione} \\ & \text{in cui } y = 0 \\ f(\underline{x}, y+1) = h(\underline{x}, y, f(\underline{x}, y)) & \text{Passo induttivo. Se vale} \\ & \text{qui, vale per ogni } n \end{cases}$$

dove per \underline{x} si intende il vettore degli input, y escluso.

Vediamo subito un esempio che serva a chiarire il funzionamento di questo, all'apparenza complicato, costrutto funzionale. Cerchiamo di implementare il prodotto di due naturali x ed y tramite questo meccanismo. Per prima cosa, vediamo di dare una definizione ricorsiva del prodotto, e quindi, da questa, determinare lo schema ricorsivo di cui sopra:

$$x^y = \begin{cases} x^0 = 0 & \text{Base dell'induzione} \\ x^{(y+1)} = (x^y) + x & \text{Passo induttivo} \end{cases} \longrightarrow \begin{cases} g(x) = x^0 = 0 \\ h(x, y, z) = \cancel{x^y} \cancel{+ z} = z + x \end{cases}$$

Figura 4.2

sopra, per brevita' abbiamo posto z al posto di $f(x, y)$ nella definizione di h , come nel testo si fa, per convenzione, quasi sempre. Dunque, per dimostrare che una certa funzione si puo' implementare tramite ricorsione primitiva, basta ricavare, da uno schema tipo quello di figura 4.2, le funzioni g, h , e quindi implementarle tramite costrutti funzionali. Ad esempio, nel caso sopra avro'

- $g(x) =_{\text{def}} 0$
- $h(x, y, z) =_{\text{def}} ((P^3_{3,3} \wedge P^3_{1,1}) ; +)$ (che equivale alla somma $x+z$)

Diamo un esempio di esecuzione del programma:

$$3^2 = h(3, 1, f(3, 1)) = 3 + h(3, 0, f(3, 0)) = 3 + (3 + f(3, 0)) = 3 + (3 + g(3)) = 3 + (3 + 0) = 3 + 3 = 6$$

Dall'esecuzione del programma qui sopra si capisce come funziona la ricorsione primitiva al suo interno. In pratica viene eseguita $h(x, y, z)$ fino a che z non diventa il caso base (e cioe' $f(\underline{x}, 0) = g(\underline{x})$). Quindi, vengono risolte tutte le formule ricorsive che erano rimaste "pendenti", fino a risolvere $f(\underline{x}, y)$, che sara' il nostro risultato finale.

4.4 Generare l'esponenziazione tramite ricorsione primitiva

Cominciamo qui la prima delle due parti di una dimostrazione che ci portera' a dire che la ricorsione primitiva e' equivalente all'esponenziazione. Cominciamo col far vedere che lo schema generale dell'esponenziazione puo' essere definito per ricorsione primitiva. Avremo:

$$\exp(f)(\underline{x}, 0) = \underline{x} \longrightarrow \begin{cases} g(\underline{x}) = \underline{x} \\ h(\underline{x}, y, z) = f(z) \end{cases}$$

$$\exp(f)(\underline{x}, y+1) = f(\exp(f)(\underline{x}, y))$$

dove, come al solito, si e' sostituito $\exp(f)(\underline{x}, y)$ con z . In questo caso, implementare le due funzioni g ed h risulta molto agevole. Infatti abbiamo:

- $g(\underline{x}) =_{\text{def}} P^{\underline{n}}_{1,n}$
- $h(\underline{x}, y, z) =_{\text{def}} f(z)$

Quindi f viene applicata a se stessa tante volte (in modo fintizio) fino a quando non viene raggiunto il caso base. Solo a questo punto, tornando indietro con la ricorsione, f verrà effettivamente applicata ai rispettivi argomenti, fino ad ottenere, all'ultimo passaggio, l'esponenziazione completa.

4.5 Qualche esempio di ricorsione primitiva

Vediamo ora un paio di esempi in cui viene applicata la ricorsione primitiva. Cominciamo con l'implementare la seguente funzione ricorsiva:

$$\left\{ \begin{array}{l} f(x,0) = x+2 \\ f(x,y+1) = x^2 + y + f(x,y) \end{array} \right. \quad \xrightarrow{\hspace{1cm}} \quad g(x) = x + 2$$

$$h(x,y,z) = x^2 + y + z$$

Figura 4.4

da cui otteniamo le seguenti, ovvie implementazioni di g ed h :

- $g(x) =_{\text{def}} (S ; S)$
- $h(x,y,z) =_{\text{def}} (D \parallel P^2_{1,2}) ; (* \parallel +) ; (+)$

Qui, h si preoccupa di duplicare il valore di x di modo da poi poterne fare il quadrato moltiplicando le due occorrenze. Il resto della funzione si preoccupa di sommare il tutto.

Un altro esempio interessante di applicazione del costrutto di ricorsione puo' essere, data la definizione ricorsiva di $\exp(f)$ implementare la funzione somma. Per risolvere questo problema, in realta', basta prendere la definizione di figura 4.3 e le funzioni g ed h relative, e sostituire ad f la funzione Sucessore S . In questo modo la funzione somma sara' implementata senza dover ricorrere a nessun ulteriore ragionamento.

5^a Lezione

Vediamo adesso la seconda parte della dimostrazione che avevamo iniziato nel capitolo precedente. Cercheremo quindi, in questo numero, di implementare la ricorsione primitiva tramite l'esponenziazione.

5.1 Generare la ricorsione primitiva tramite l'esponenziazione.

Ora, dimostreremo che, data una funzione $f(x,y)$ definita tramite ricorsione primitiva, il procedimento ricorsivo che serve per ricavare f puo' essere simulato dal costrutto di esponenziazione.

Sappiamo che, con la ricorsione primitiva, $f(x,2)$ puo' essere vista in termini di $f(x,1)$, $f(x,1)$ in termini di $f(x,0)$ ed $f(x,0)$ in termini di $g(x)$. Tutto cio' puo' essere riassunto dalle seguenti notazioni:

$$\begin{aligned} f(x,y+1) &= f(x,1) = h(x,y) = h(x,0,f(x,0)) = h(x,0,g(x)) \\ f(x,y+1) &= f(x,2) = h(x,y) = h(x,1,f(x,1)) = h(x,1,h(x,0,f(x,0))) = h(x,1,h(x,0,g(x))) \end{aligned}$$

osservando bene questa notazione, possiamo ricavare cio' che ci siamo prefissi, ovvero, partendo dalle nostre funzioni g ed h possiamo trovare una esponenziazione adeguata a simulare il processo ricorsivo. In particolare, vediamo che h viene applicata ai suoi argomenti un numero di volte uguale al valore di y . Inoltre, vediamo che il modo in cui si sviluppa $f(x,y)$ al variare di y e' sempre lo stesso, quindi sara' facile trovare un algoritmo che simuli tale processo.

Scriviamo quindi il programma che simula la ricorsione primitiva. Di seguito lo commenteremo in modo da capirne bene il funzionamento:

$$\boxed{\text{rec}(g,h)(x,y) = (\mathbf{P}^2_{1,1} \wedge (\mathbf{P}^2_{2,1}; 0) \wedge (\mathbf{P}^2_{1,1}; g) \wedge \mathbf{P}^2_{2,2}) ; \exp(t); \mathbf{P}^3_{3,3} \quad \text{con } t = \text{def } \mathbf{P}^3_{1,1} \wedge (\mathbf{P}^3_{2,2}; S) \wedge h}$$

Vediamo di capire bene il funzionamento di questo programma. Per prima cosa, le prime tre istruzioni che precedono $\exp(t)$, hanno il compito di generare la sequenza $(x,0,g(x))$, che abbiamo visto essere la base della ricorsione. A questo punto, alla base della ricorsione bisognera' applicare h tante volte quanto e' il valore di y , che e' per questo proiettato come quarto input della funzione $\exp(t)$: esso in pratica agira' come indice del ciclo FOR.

A questo punto comincia ad agire la funzione $t : \mathbb{N}^3 \rightarrow \mathbb{N}^3$, la quale *prende in input una terna* (x,y,z) e *restituisce in output la terna successiva* $(x,y+1,h(x,y,z)) = (x,y+1,f(x,y))$.

Al termine di questo procedimento, il risultato si trovera' nella terza posizione posizione della terna di cui sopra, e sara' la nostra $f(x,y)$ che cercavamo.

Abbiamo cosi' dimostrato, abbinando questa dimostrazione a quella vista nel paragrafo 4.4, che la ricorsione primitiva e l'esponenziazione sono due costrutti equivalenti, in quanto da uno di questi e' possibile ricavare l'altro.

Adesso passeremo ad analizzare altri costrutti che ci saranno molto utili in futuro, pensando sempre che il nostro obiettivo sia quello di creare un linguaggio funzionale che sia in grado di implementare una qualsiasi funzione calcolabile. Per arrivare a questo risultato, come si puo' intuire, ci mancano ancora due costrutti fondamentali, ovvero l'IF-THEN e il WHILE-DO.

Prima di tutto questo, pero' vediamo ancora qualche esempio di applicazione del costrutto di ricorsione primitiva.

5.2 Definizione della somma per ricorsione primitiva

Nella figura sottostante, vediamo subito come la somma possa essere implementata tramite ricorsione primitiva. Vediamo quindi come la somma possa essere pensata ricorsivamente:

Figura 5.1

$$x+y = \begin{cases} f(x,0) = x+0 = x \\ f(x,y+1) = x+(y+1) = (x+y)+1 \end{cases} \longrightarrow \begin{cases} g(x) = x \\ h(x,y,z) = z+1 \end{cases}$$

da cui ricaviamo le seguenti, ovvie implementazioni per g ed h :

- * $g(\underline{x}) =_{\text{def}} P^1_{1,1}$
- * $h(\underline{x},\underline{y},\underline{z}) =_{\text{def}} ((P^3_{3,3}) \parallel (0;S)) ; +$

Ora, vediamo una esecuzione di questo programma:

$$\boxed{3+2 = h(3,1,f(3,1)) = 1 + h(3,0,f(3,0)) = 1 + (1 + f(3,0)) = 1 + (1 + 3) = 5}$$

5.3 Definizione del predecessore di un numero

Vediamo ora come implementare la **funzione predecessore** ($x - 1$, con un puntino sul meno). Notiamo che, qualora $x = 0$, $x-1$ andrebbe nei numeri negativi, ma noi lavoriamo sui naturali, per cui ponremo $0-1 = 0$.

Cominciamo col vedere una soluzione tradizionale. L'idea, per calcolare $x-1$, e' quella di inizializzare due celle di memoria a zero, una delle quali rappresentera' il valore di x , l'altra il valore di $(x-1)$. In ogni modo, guardando il programma tutti i concetti appariranno subito piu' chiari:

$$\boxed{-1(A1)(A2) = (0 \parallel 0 \parallel P^1_{1,1}) ; \exp(t) ; P^2_{1,1} \quad \text{con } t =_{\text{def}} P^2_{2,2} \wedge (P^2_{2,2} ; S)}$$

Quindi le due celle di memoria assumeranno I valori $(0,1), (1,2), (2,3), \dots, (x-1,x)$

Vediamo un'altra soluzione allo stesso problema, questa volta pero' utilizzando la ricorsione primitiva. Si noti che, contrariamente a tutti gli esempi sulla ricorsione fatti in precedenza, in questo caso l'input in ingresso e' composto da un solo numero naturale. In questi casi, l'unico input da considerare viene posto nella variabile y , mentre il vettore \underline{x} , avra' lunghezza nulla. Quindi avremo $f(y)$ ed $h(y,z)$ al posto di $f(x,y)$ ed $h(x,y,z)$. Cominciamo col vedere lo schema ricorsivo relativo a questo problema:

Figura 5.2

$$\boxed{\begin{cases} 0-1 = 0 \\ (y+1)-1 = y \end{cases} \longrightarrow \begin{cases} g(y) = 0 \\ h(y,z) = y \end{cases}}$$

tralasciamo l'implementazione delle funzioni g ed h , vista la sua estrema semplicita'. Un esempio di esecuzione del programma e', ad esempio,

$$\boxed{(y-1)(4) = h(3,f(3)) = 3}$$

5.4 Definizione della sottrazione per ricorsione primitiva

Vediamo ora come implementare la sottrazione tra naturali per ricorsione primitiva. Ovviamente, non avendo a disposizione I numeri negativi, $x - y = 0$ se $y > x$:

$$x-y = \begin{cases} x-0 = x \\ x-(y+1) = (x-y)-1 \end{cases} \longrightarrow \begin{cases} g(x) = x \\ h(x,y,z) = z-1 \end{cases}$$

Figura 5.3

un esempio di esecuzione di questo programma potrebbe essere:

$$(5-3) = h(5,2,f(5,2)) = h(5,1,f(5,1)) - 1 = (h(5,0,f(5,0))-1)-1 = ((f(5,0)-1)-1)-1 = 5-1-1-1 = 2$$

5.5 Definizione di Sg(segno) e !Sg(segno negato) per ricorsione primitiva

Vediamo ora le definizioni ricorsive di segno e segno negato, seguite dalle loro implementazioni tramite ricorsione primitiva:

Figura 5.4

$$\begin{aligned} Sg(y) &= \begin{cases} 0 & \text{se } y = 0 \\ 1 & \text{se } y < 0 \end{cases} \xrightarrow{\hspace{1cm}} \begin{cases} Sg(0) = 0 \\ Sg(y+1) = 1 \end{cases} \xrightarrow{\hspace{1cm}} \begin{cases} g(y) = 0 \\ h(y,z) = 1 \end{cases} \quad \overline{Sg}(y) = \begin{cases} 1 & \text{se } y = 0 \\ 0 & \text{se } y < 0 \end{cases} \end{aligned}$$

abbiamo volutamente tralasciato le definizioni di g ed h per !Sg, in quanto sono facilmente desumibili da quelle trovate per Sg. Anche l'implementazione di tali funzioni, essendo molto semplice, non viene qui sviluppata.

5.6 Implementazione della funzione min(x,y)

Possiamo risolvere il problema di trovare il min(x,y) tramite il seguente algoritmo:

$$\min(x,y) = y * Sg(x-y) + x * (!Sg(x-y))$$

Questo algoritmo potra' essere implementato dal seguente programma

$$Min(\cancel{x},\cancel{y}) = [P^2_{2,2} \wedge (- ; Sg) P^2_{1,1} \wedge (- ; !Sg)] ; (* \parallel *) ; +$$

5.7 Implementazione del costrutto IF-THEN

Un modo per implementare il costrutto IF – THEN puo' essere quello di implementare una certa funzione h(x,y), che assuma valore f(x) se y=0, e g(x) se y<>0. Una funzione del genere puo' essere generata da questo semplice algoritmo:

$$h(x,y) = f(x) * (!Sg(y)) + g(x) * Sg(y)$$

Tralasciamo l'implementazione di questo algoritmo, che consiste in un programma molto semplice estremamente simile a quello appena visto per l'implementazione di min(x,y).

Vediamo ora un altro programma che genera il costrutto IF – THEN, questa volta, pero' attraverso il meccanismo dell'esponenziazione, sempre cercando di generare la funzione h(x,y) di cui sopra:

$$h(x,y) = (P^2_{1,1} \wedge (P^2_{2,2} ; !Sg) \wedge (P^2_{2,2} ; Sg)) ; (\exp(f) \parallel P^1_{1,1}) ; \exp(g)$$

Questo programma, in definitiva, puo' essere visto suddiviso in 3 macroistruzioni. La prima si preoccupa di creare la sequenza (x, !Sg(y), Sg(y)). La seconda prende x e gli applica f per !Sg(y) volte, ottenendo cosi', a seconda del numero di cicli effettuati, x o f(x). La terza prende questo risultato e gli applica g(x) Sg(y) volte, ottenendo cosi' f(x) se y=0, g(x) altrimenti. Vediamo come si comporta il programma in due casi specifici:

$h(x,2) \rightarrow x, 0, 1 \rightarrow x, 1 \rightarrow g(x)$
$h(x,0) \rightarrow x, 1, 0 \rightarrow f(x), 0 \rightarrow f(x)$