CALCOLI SU STRINGHE DI SIMBOLI

Verranno adesso introdotti due ulteriori linguaggi di programmazione o, meglio:

- una famiglia di linguaggi per la manipolazione di stringhe su un alfabeto di n simboli, un linguaggio per ogni n;
- un linguaggio per descrivere il funzionamento di una macchina di Turing mediante istruzioni.

I primi saranno strettamente modellati su $\mathcal{S}$, il secondo sarà il più possibile simile ad esso.

La necessità di cambiare istruzioni è imposta dal fatto che quelle che sono istruzioni elementari e del tutto naturali nel contesto degli interi sono prive di significato diretto e devono opportune interpretazioni artificiose come istruzioni per manipolare stringhe

Sia adesso A un alfabeto di n simboli, $A = \{s_1, \ldots, s_n\}$ e A^* è l'insieme di tutte le stringhe su A compresa la stringa vuota 0 (in termini algebrici A^* è il monoide generato da A).

Per ciascun $n > 0$ verrà quindi introdotto un linguaggio di programmazione $\mathcal{S}_n$.

Tutte le regole sintattiche e le convenzioni per ciascun $\mathcal{S}_n$ sono del tutto analoghe a quelle di $\mathcal{S}$ e non verranno perciò esplicitamente enunciate.

CALCOLI SU STRINGHE DI SIMBOLI

Verranno adesso introdotti due ulteriori linguaggi di programmazione o, meglio:

- una famiglia di linguaggi per la manipolazione di stringhe su un alfabeto di n simboli, un linguaggio per ogni n;
- un linguaggio per descrivere il funzionamento di una macchina di Turing mediante istruzioni.

I primi saranno strettamente modellati su $\mathcal{S}$, il secondo sarà il più possibile simile ad esso.

La necessità di cambiare istruzioni è imposta dal fatto che quelle che sono istruzioni elementari e del tutto naturali nel contesto degli interi sono prive di significato diretto e dopo opportune interpretazioni artificiose come istruzioni per manipolare stringhe

Sia adesso A un alfabeto di n simboli, $A=\{s_1,\dots,s_n\}$ e A^* è l'insieme di tutte le stringhe su A compresa la stringa vuota 0 (in termini algebrici A^* è il monoide generato da A).

Per ciascun $n>0$ verrà quindi introdotto un linguaggio di programmazione $\mathcal{S}_n$.

Tutte le regole sintattiche e le convenzioni per ciascun $\mathcal{S}_n$ sono del tutto analoghe a quelle di $\mathcal{S}$ e non verranno perciò esplicitamente enunciate.

ISTRUZIONI DEI LINGUAGGI $\mathcal{S}_n$

$V \leftarrow \sigma V$ un'istruzione per ciascun simbolo dell'alfabeto A	concatena il simbolo σ alla sinistra della stringa che rappresenta il valore di V
$V \leftarrow V^-$	cancella il simbolo finale della stringa che rappresenta il valore di V se V=0 non fare niente
IF V ENDS σ GOTO L un'istruzione per ciascun simbolo dell'alfabeto A e per ogni etichetta L	se la stringa che rappresenta il valore di V termina col simbolo σ, esegui la prima istruzione etichettata L; altrimenti esegui l'istruzione successiva

Analogamente al caso di $\mathcal{S}$, diremo che una funzione parziale di m variabili su A^* è parzialmente calcolabile in $\mathcal{S}_n$ se esiste un programma in $\mathcal{S}_n$ che la calcola.

Osservazione

Possiamo pensare ai programmi di $\mathcal{S}_n$, per qualche n, o come a pure manipolazioni di stringhe su un alfabeto di n simboli o come a calcoli su numeri che sono rappresentati in base n.
Questo punto di vista è utile quando si vogliono paragonare le capacità di calcolo dei vari $\mathcal{S}_n$ tra loro e con quella di $\mathcal{S}$.

Per poter fare un confronto tra i linguaggi $\mathcal{S}$ ed $\mathcal{S}_n$ è necessario mostrare la calcolabilità in ciascun $\mathcal{S}_n$ delle due funzioni $x+1$ e $x \dot{-} 1$, questa verrà a sua volta semplificata dalla possibilità di usare alcune macro opportune.

IF V≠0 GOTO L

Espansione:

IF V ENDS S_1 GOTO L
IF V ENDS S_2 GOTO L
. . . . . .
IF V ENDS S_n GOTO L

V←0

Espansione:

[A] V←V⁻
IF V≠0 GOTO A

(Idea: cancella l'ultimo simbolo e continua il procedimento finché non avrai cancellato tutti i simboli)

GOTO L

Espansione:

Z←0
Z←S_1Z
IF Z ENDS S_1 GOTO L

Il blocco di istruzioni:

IF V ENDS S_1 GOTO B_1
IF V ENDS S_2 GOTO B_2
. . . . . .
IF V ENDS S_n GOTO B_n

si chiama FILTRO e può essere abbreviato in IF V ENDS S_i GOTO B_i $(1 \leq i \leq n)$

$V' \leftarrow V$

Espansione:

```
1        Z ← 0
2        V' ← 0
3    [A] IF V ENDS Si GOTO Bi   (1≤i≤n)  (FILTRO)
4        GOTO C
5    [Bi] V ← V⁻                 }
6        V' ← Si V'              }  i = 1, 2, ..., n
7        Z ← Si Z                }
8        GOTO A                  }
9    [C] IF Z ENDS Si GOTO Di (1≤i≤n)  (FILTRO)
10       GOTO E
11   [Di] Z ← Z⁻                 }
12       V ← Si V                }  i = 1, 2, ..., n
13       GOTO C                  }
```

Osservazioni

- Le istruzioni 3 e 9 sono dei filtri e quindi, in realtà blocchi di n istruzioni ciascuno.
- Anche i blocchi di istruzioni 5-8 e 11-13 dovrebbero in realtà essere ripetuti n volte.
- Il programma si basa nel funzionamento successivo dei due cicli $A \rightleftarrows B$ e $C \rightleftarrows D$ (dove B e D indicano, genericamente, quel B_i o D_i che di volta in volta viene chiamato).
  - Il primo ciclo cancella ad ogni passo una lettera di V che ricopia sia in V' sia in Z (che sono state entrambe azzerate)
    Alla fine di questo processo V è cancellata e due sue copie sono presenti in V' e in Z.
  - Il secondo ciclo con un procedimento simile ricopia Z in V.
    Alla fine avremo dunque $Z = 0$ e sia V che V' col valore iniziale di V.

Abbiamo adesso gli strumenti per presentare due programmi semplici che calcolano le funzioni $x+1$ e $x \dot{-} 1$ in $\mathcal{S}_n$.

Prima di fare questo però è forse opportuno, anche se non necessario, riprendere il problema della rappresentazione dei numeri in A^*.

Ciò che ci serve è una rappresentazione degli interi uno-a-uno. Quindi la solita rappresentazione decimale che consente ambiguità del tipo

$$007 = 7$$

non è utile da questo punto di vista.

Le rappresentazioni che ci interessano quindi sono del tutto simili alle solite tranne per il punto precedente il che comporterà che in una rappresentazione in base n i simboli elementari rappresenteranno i numeri da 1 a n anziché da 0 ad $n-1$ e lo 0 verrà, a sua volta, identificato con la stringa vuota che indicheremo con lo stesso simbolo 0.

Sia dunque un alfabeto A di n lettere $s_1, \ldots, s_n$. Assumeremo che gli s_i sono stati posti in un ordine definito (che è quello dato dagli indici) e ci riferiremo sempre a questo ordine.

Consideriamo adesso la stringa:

$$w = s_{i_k} s_{i_{k-1}} \dots s_{i_1} s_{i_0}$$

associamo a w il numero intero:

$$(*) \qquad x = i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \dots + i_1 \cdot n + i_0$$

w sarà allora chiamata la notazione in base n (in A) del numero definito da $(*)$

Osservazione.
Mentre la solita notazione in base n che ammette lo 0, ha bisogno di almeno due simboli, la notazione presente funziona anche con un solo simbolo, nel qual caso un numero n è rappresentato da una stringa eguale alla concatenazione di n copie del nostro unico simbolo a disposizione.
Si mostra che tale rappresentazione è unica nel senso che dato il numero x possiamo recuperare gli indici i_ℓ e quindi la stringa w.
Si può anche mostrare che questo può essere fatto usando delle funzioni ricorsive primitive.
Esisterà quindi una funzione ricorsiva primitiva $h(m, n, x)$ (dove x è il numero del quale vogliamo recuperare la stringa corrispondente, n è la cardinalità dell'alfabeto A ed m l'indice cercato) tale che:

$$i_m = h(m, n, x) \qquad \text{per } m = 0, 1, \dots, k.$$

Sia $A = \{s_1, \ldots, s_n\}$

consideriamo la stringa:

$$w = s_{i_k} s_{i_{k-1}} \cdots s_{i_1} s_{i_0}$$

Questa è la rappresentazione in base n del numero:

$$x = i_k \cdot n^k + i_{k-1} n^{k-1} + \ldots + i_1 n + i_0$$

Quindi:

- la base è data dalla cardinalità dell'alfabeto
- l'esponente al quale devo elevare n nei vari luoghi dipende dalla posizione del simbolo considerato nella stringa w
- il coefficiente dalla posizione del simbolo nell'ordinamento dato.

LO ZERO È RAPPRESENTATO DALLA STRINGA VUOTA!

Esempi

$$A = \{ \underset{x}{s_1}, \underset{y}{s_2}, \underset{z}{s_3} \} \qquad \text{Card}(A) = 3$$

$$w = \underset{z}{s_3}\ \underset{z}{s_3}\ \underset{x}{s_1}\ \underset{y}{s_2}$$

$$x = 3 \cdot 3^3 + 3 \cdot 3^2 + 1 \cdot 3 + 2$$

Modifica della base decimale

$$A = \{1, 2, \ldots, 9, D\}$$

Esempi:

stringa : 763

numero associato: $7 \cdot 10^2 + 6 \cdot 10 + 3 = 763$

stringa : 3D

numero associato : $3 \cdot 10 + 10 = 40$

A questo punto avremmo gli elementi per introdurre una serie di funzioni su A^* e dimostrare la loro ricorsività primitiva. Queste funzioni ci serviranno in seguito. Per il momento preferiamo ritornare all'argomento che avevamo lasciato in sospeso e cioè la costruzione di programmi che calcolano $x+1$ e $x \dot{-} 1$ in $\mathcal{S}_u$.

I due programmi come vedremo tra poco sono estremamente semplici e si capisce immediatamente il meccanismo che sta alla loro base.
L'unico punto da tener presente (e che è quello che genera due cicli diversi nei programmi è il problema del riporto.

Ricordiamo che nel caso dell'addizione, in rappresent. decimale, se sommiamo 1 ad un numero di più cifre che finisce per 9, avremo un riporto che si propagherà fino a quando non incontreremo una cifra inferiore a 9 che lo riassorbirà.
ANALOGAMENTE NELLA NOSTRA RAPPRESENTAZIONE DEI NUMERI AVREMO UN PROBLEMA DI RIPORTO RISPETTIVAMENTE QUANDO VORREMO AGGIUNGERE 1 AD UN NUMERO LA CUI RAPPRESENTAZIONE PRESENTA COME ULTIMA CIFRA L'ULTIMO SIMBOLO NEL NOSTRO ORDINAMENTO E QUANDO VORREMO SOTTRARRE 1 AD UN NUMERO LA CUI RAPPRESENTAZIONE PRESENTA COME ULTIMA CIFRA IL PRIMO SIMBOLO DI A NEL NOSTRO ORDINAMENTO.

```
[B]    IF X ENDS s_i GOTO A_i      (1 ≤ i ≤ n)
       Y ← s_1 Y
       GOTO E
[A_i]  X ← X⁻
       Y ← s_{i+1} Y               1 ≤ i < n
       GOTO C
[A_n]  X ← X⁻
       Y ← s_1 Y
       GOTO B
[C]    IF X ENDS s_i GOTO D_i      (1 ≤ i ≤ n)
       GOTO E
[D_i]  X ← X⁻
       Y ← s_i Y                   1 ≤ i ≤ n
       GOTO C
```

il caso $i = n$ è trattato a parte.

Fig. 2.3. Program which computes $x + 1$ in $\mathscr{S}_n$.

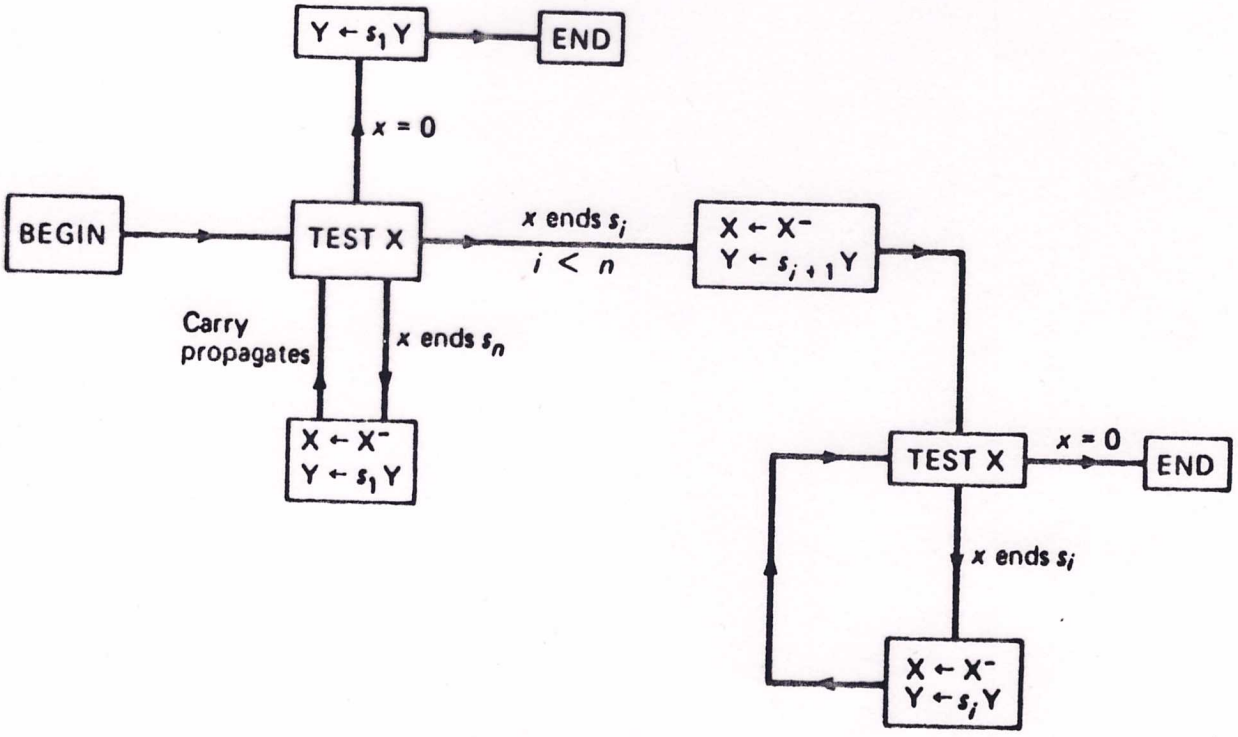

Fig. 2.2. Flow chart for computing $x + 1$ in $\mathscr{S}_n$.

```
[B]    IF X ENDS s_i GOTO A_i      (1 ≤ i ≤ n)
       GOTO E
[A_i]  X ← X⁻
       Y ← s_{i-1} Y               } 1 < i ≤ n
       GOTO C
[A_1]  X ← X⁻
       IF X ≠ 0 GOTO C_2
       GOTO E
[C_2]  Y ← s_n Y
       GOTO B
[C]    IF X ENDS s_i GOTO D_i      (1 ≤ i ≤ n)
       GOTO E
[D_i]  X ← X⁻
       Y ← s_i Y                   } 1 ≤ i ≤ n
       GOTO C
```

il caso $i = 1$ è trattato a parte.

Fig. 2.5. Program which computes $x \dot{-} 1$ in $\mathcal{S}_n$.

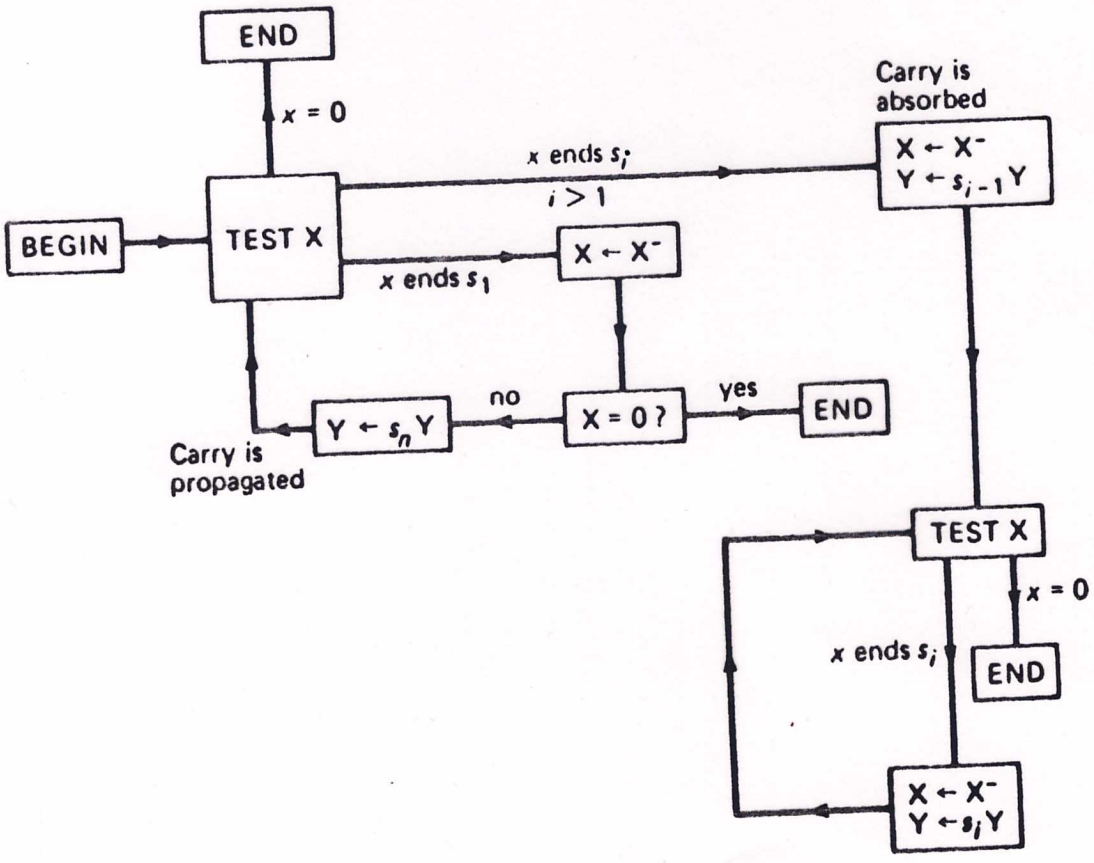

Fig. 2.4. Flow chart for computing $x \dot{-} 1$ in $\mathcal{S}_n$.

Poniamo a questo punto presentare i primi risultati del confronto della capacità di calcolo dei vari $\mathcal{S}$ e $\mathcal{S}_n$.

Ricordiamo che il punto di vista assunto per poter fare il confronto è che tutti i calcoli sono calcoli numerici e le stringhe sono solo un modo di rappresentare i numeri in una base eguale alla cardinalità dell'alfabeto.

Proposizione

Una funzione è parzialmente calcolabile (in $\mathcal{S}$) se e solo se è parzialmente calcolabile in $\mathcal{S}_1$.

Dimostrazione

Se guardiamo ad $\mathcal{S}$ e ad $\mathcal{S}_1$ dal punto di vista "numerico", i due linguaggi si comportano nello stesso identico modo data la corrispondenza esistente tra le istruzioni corrispondenti

(*) $V \leftarrow S_1 V \quad \longleftrightarrow \quad V \leftarrow V+1$.

(**) $V \leftarrow V^- \quad \longleftrightarrow \quad V \leftarrow V-1$

Le (**) diminuiscono entrambe, nei rispettivi linguaggi, di una unità il valore di V.
Analogamente, le (*) aumentano di una unità il valore di V.

Le istruzioni di salto sono anch'esse identiche perché "V ENDS S_1", essendoci un solo simbolo, è equivalente a dire che $V \neq 0$.

I due linguaggi sono quindi identici.

Mostriamo adesso che:

Proposizione Se una funzione è parzialmente calcolabile, allora è parzialmente calcolabile in S_n per ogni n.

Dimostrazione

Sia P il programma nel linguaggio S che calcola f.
Trasformiamo il programma P in S in un programma P_{S_n} rimpiazzando le istruzioni di P con corrispondenti macro in S_n nel modo seguente:

	in S		in S_n
Istruz.:	$V \leftarrow V+1$	$\longrightarrow$ macro:	$V \leftarrow V+1$
"	$V \leftarrow V-1$	$\longrightarrow$ "	$V \leftarrow V \dot{-} 1$
"	IF $V \neq 0$ GOTO L	$\longrightarrow$ "	IF $V \neq 0$ GOTO L

Il nuovo programma P_{S_n} in S_n sarà molto più lungo di P, una volta sostituita ogni macro dalla sua macroespansione, ma ovviamente calcolerà in S_n la stessa funzione che P calcolava in S, perché non fa altro che simulare P in S_n.

Passiamo adesso ad introdurre il linguaggio di POST-TURING

Questo è un modo di esprimere il funzionamento di una macchina di Turing, come noi l'abbiamo già introdotta, usando istruzioni del tipo incontrato finora e programmi come liste di istruzioni.

Si rinvia alla discussione già fatta per tutte le motivazioni e dettagli di altro genere.

Passiamo alla presentazione del nostro linguaggio:

IL LINGUAGGIO DI POST-TURING Σ

ISTRUZIONE	INTERPRETAZIONE
PRINT σ	RIMPIAZZA CON σ IL SIMBOLO PRESENTE SUL QUADRATO ESAMINATO
IF σ GOTO L	VAI ALLA PRIMA ISTRUZIONE ETICHETTATA L SE IL SIMBOLO ESAMINATO È σ ALTRIMENTI PASSA ALL'ISTRUZIONE SUCCESSIVA
RIGHT	ESAMINA IL QUADRATO IMMEDIATAMENTE A DESTRA DI QUELLO ESAMINATO
LEFT	ESAMINA IL QUADRATO IMMEDIATAMENTE A SINISTRA DI QUELLO ESAMINATO