LISP e Calcolabilità

Eugen Neidl

Il LISP è un linguaggio di programmazione, la calcolabilità è la teoria delle funzioni che sono calcolabili mediante procedimenti meccanici.

L'uno e l'altra godono fama di essere difficili e qualche volte anche un po' noiosi. Di entrambi, però, si possono dare delle presentazioni divertenti e si può evitare di affrontare le difficoltà maggiori pur continuando a dare una idea essenziale di ciò di cui si sta parlando.

Nell'articolo, che vuole essere una rapida introduzione a questo linguaggio di programmazione, vorremmo anche mostrare come il LISP e la calcolabilità possano essere messi in relazione in modo tale che la conoscenza dell'uno aiuti lo studio dell'altra e viceversa.

1. Un po' di storia

Il LISP è stato creato nel 1958 da John McCarthy per lavori nel campo dell'Intelligenza Artificiale. Il suo nome è un acronimo di *LISt Processing*, elaborazione di liste. Però, curiosamente, nel titolo del suo articolo apparso nel 1960 sulla rivista Communications of A.C.M.: "Mathematical Functions of Symbolic Expressions and their Computation by Machine" non si parla di Intelligenza Artificiale ma, proprio di funzioni e della loro calcolabilità.

Nei quasi quaranta anni della sua esistenza questo linguaggio si è arricchito di vari dialetti e oggi si può parlare di almeno tre generazioni di LISP. Dopo il FORTRAN, è il linguaggio di programmazione più vecchio.

La calcolabilità è una teoria che ha una vita molto più lunga, comincia almeno con i lavori di Euclide. Il suo metodo per calcolare il massimo comun divisore è ritenuto uno dei più antichi algoritmi, dove per algoritmo intendiamo una descrizione precisa per calcolare una funzione matematica.

Nel nostro secolo, la calcolabilità è molto legata ai lavori, degli anni '30, di Gödel, Turing, Church, Kleene e Post, per citare alcuni nomi In un momento in cui non esistevano ancora i computer, le macchine erano virtuali. Oggi l'informatica è una scienza ben distinta dalla matematica, il computer è onnipresente, i bambini sono in grado di usarlo con maggiore facilità degli adulti e in genere si pensa che si può calcolare tutto. Far capire alla gente che già prima della esistenza del computer, più di sessanta anni fa, è stato dimostrato che non tutto possa essere calcolato, neanche con delle macchine superpotenti ancora da costruire, è un po' difficile: arriva come un choc e come una certa incredulità nella mente.

Per ogni studente universitario di informatica è già previsto seguire un corso di calcolabilità, per un allievo del liceo o delle scuole medie che segue un corso di programmazione sarà un ottimo confronto con i limiti del computer.

2. Iniziamo il LISP con l'aritmetica

2.1 Sintassi

Il LISP usa la notazione polacca prefissa completamente parentesizzata:

- Il primo elemento di ogni formula è un simbolo, che ci dice quale operazione vogliamo compiere, ed è seguito da tutti gli operandi,

- il tutto è circondato da parentesi.

In sintesi: la sintassi ha bisogno solo di parentesi e di spazi.

Esempio:
L'espressione aritmetica 34 + 209 si scrive in LISP come (+ 34 209). L'operatore è l'usuale simbolo "+" della matematica, gli operandi sono i numeri 34 e 209. L'uso delle parentesi ci permette di evitare di dover definire delle regole di precedenza per i vari operatori e, quindi, di effettuare delle combinazioni in modo molto facile.

Altri esempi:
34 + 45 * 7 si scrive in LISP come (+ 34 (* 45 7))
(34 + 45) * 7 si scrive in LISP come (* (+ 34 45) 7)
La prima espressione è la somma di un numero e di un prodotto di numeri,

la seconda è il prodotto di una somma di numeri e di un numero.

Questo linguaggio è stato accompagnato fin dalla sua nascita anche da una polemica costante: viene detto che la notazione è brutta, illeggibile, incomprensibile e, infine, che si devono usare troppe parentesi. Quest'ultima critica ha portato a fornire - scherzosamente - un'altra spiegazione del nome LISP che sarebbe un acronimo della definizione *vera* del linguaggio: a **L**ot of **I**nsipid **S**tupid **P**arentheses. Questa definizione ironica ricorda un fatto vero: moltissimi principianti hanno sbattuto la testa alla ricerca di errori nei loro programmi che nascevano solo da parentesi sbagliate. Chiedo al lettore solo un piccolo sforzo intellettuale per acquistare familiarità con questa sintassi. I vantaggi che essa presenta saranno evidenti nel seguito.

2.2 Uso al computer

Si dice che il LISP è un linguaggio interattivo, con il LISP il computer si usa come un calcolatore tascabile. Subito dopo aver digitato una formula alla tastiera, si ottiene il risultato e la macchina è di nuova pronta per rispondere a un altro quesito.

Esempi: (le cose stampate dal computer sono in grassetto):

? (+ 34 209)
= 243

? (+ 34 (* 45 7))
= 349

? (* (+ 34 45) 7)
= 553

Quando il computer è pronto per una formula, stampa un punto interrogativo. Si chiama un *"prompt"*. Prima del risultato stampa un "=". Dopo di che è di nuovo pronto per un' altra domanda.

2.3 Altri operatori aritmetici

Ecco i simboli usati per le quattro operazione aritmetiche:

+	Somma
-	Differenza
*	Prodotto
/	Quoziente (in virgola mobile, ciò anche per numeri reali).

Con questi operatori si possono ottenere formule e loro combinazioni di una complessità qualsiasi.

Esempi:

? (* 23 70)
= 1610

? (/ 978 25)
= 39.12

Per indicare un'operazione non si deve necessariamente usare un solo carattere disponibile sulla tastiera del computer. Si possono usare anche parole intere (spesso quelle già introdotte hanno nome inglese), che indicano altre operazioni o funzioni. Per esempio, la divisione tra numeri interi è indicata con la parola inglese QUOTIENT, e il resto della divisione intera con la parola MODULO.

Esempi:

? (QUOTIENT 978 25)
= 39

? (MODULO 978 25)
= 3

Fino a questo punto non si vede nessuna differenza essenziale con altri linguaggi di programmazione e in effetti non ci sarebbe nessuna differenza se non avessimo la possibilità di definire nuove funzioni e operatori in modo estremamente semplice e immediato.

2.4 Definizione di funzioni

Per definire una funzione abbiamo bisogno di tre cose opportunamente combinate tra loro: il nome della funzione, la lista dei parametri e una espressione *definienda* la quale fa uso di altre funzioni. Si da un nome alle espressioni che vengono usate spesso.

Esempio in notazione matematica per definire la funzione quadrato:

```
quad :=  x --> x*x
```

"quad" è il nome scelto per la nuova funzione da definire, "x" è il parametro della funzione e "x*x" è l'espressione definienda che fa uso del prodotto.

I simboli ":=" e "-->" sono quelli usualmente utilizzati nelle definizioni.

Vediamo adesso lo stesso esempio in LISP

```
? (DE QUAD (X) (* X X))
= QUAD
```

Il nuovo elemento che abbiamo introdotto è il costruttore "DE" che indica al sistema che stiamo dando una definizione. Dopo seguono le tre parti, naturalmente in notazione polacca prefissa.

Il risultato di una definizione è il nome della funzione definita.

```
 (DE QUAD (X) (* X X))
  ^      ^       ^      ^
  |      |       |      |----------< espressione definienda (qui si usa il
prodotto)
  |      |       |-----------------< lista dei parametri (qui un solo X)
  |            |-------------------------< nome della funzione da definire (qui
QUAD)
  |---------------------------------< il costruttore di definizioni DE
```

Dopo la definizione, la funzione QUAD può essere usata come qualsiasi altra funzione già definita, per esempio:

```
? (QUAD 5)
= 25
```

? (QUAD (QUAD 5))
= 625

Essa si può usare anche per definire una nuova funzione:

? (DE CUBO (X) (* X (QUAD X)))
= CUBO

? (CUBO 5)
= 125

Finora abbiamo definito due funzioni, QUAD e CUBO con il costrutto DE. Questi nomi sono aggiunti a quelli che il sistema conosce già e possono essere utilizzati come un operatore qualsiasi.

2.5 Definizioni per casi

Spesso una nuova funzione non è definita mediante una sola espressione ma mediante due o anche più espressioni, a seconda delle condizioni che sono verificate.

Esempio del valore assoluto:

$$|x| := \begin{cases} x & \text{per } x>0 \\ -x & \text{altrimenti} \end{cases}$$

La condizione utilizzata in questa definizione è il confronto con zero. Il dominio è diviso in due parti: i numeri superiori a zero e gli altri. C'è il predicato "x>0" che è vero per gli elementi di una parte del dominio e falso per gli elementi dell' altra parte.

A questa situazione corrisponde una sintassi LISP. Al posto della parentesi graffa si usa una parole chiave inglese "IF" che ci permette di costruire una formula parentesizzata.

Sintassi:

(IF predicato espressione-vero espressione-falso)

Quando il predicato è vero, il valore assunto da questa formula sarà quello di espressione-vero altrimenti quello di espressione-falso.
Il nostro esempio del valore assoluto si scrive in LISP:

(IF (> X 0) X (- X))

Si osservi la sintassi polacca prefissa per il predicato.

Qui sopra abbiamo scritto in LISP la condizione che definisce il valore assoluto ma non abbiamo dato una definizione completa. Per fare ciò dobbiamo innanzitutto scegliere un nome per la funzione valore assoluto. Non possiamo utilizzare, nel solito modo, le sbarre verticali | | , per via della sintassi prefissa.

Propongo il nome "VALABS", avremo così:

? (DE VALABS (X) (IF (> X 0) X (- X)))
= VALABS

? (VALABS 4)
= 4

? (VALABS -5)
= 5

I mezzi introdotti finora ci permettono di definire ogni funzione aritmetica calcolabile.

Per rinfrescare la memoria, elenchiamo ciò che abbiamo introdotto finora:
- funzioni di base: +, -, *, /, QUOTIENT, MODULO
- predicati di base: <, >, =
- espressione alternativa: (IF PREDICATO VERO FALSO)
- definizione di funzioni: (DE NOME PARAMETRI ESPRESSIONE)

Spesso, una definizione LISP è solo una traduzione della notazione matematica in notazione polacca prefissa.

Vediamo adesso degli esempi classici di funzioni ricorsive.

2.6 Ricorsività

L'elevazione di un numero a una potenza di un numero naturale si può definire in modo ricorsivo, usando il prodotto.

Esempio:

$$X^N := \begin{cases} 1 & \text{per } N=0 \\ X^{N-1} * X & \text{altrimenti} \end{cases}$$

Per tradurre questa definizione in LISP, abbiamo in primo luogo bisogno di un nome per questa funzione, la chiameremo EXPN.
È una definizione con due casi. Il predicato è il confronto con 0, il risultato nel caso vero è già un numero, il risultato nel caso falso è un prodotto uno dei cui fattori è a sua volta un' altra elevazione a potenza, cioè abbiamo bisogno di nuovo della funzione EXPN che stiamo definendo. È per questo motivo che definizioni di questo tipo si chiamano ricorsive.

In LISP si scrive dunque:

```
? (DE EXPN (X N)
?   (IF (= N 0)
?       1
?       (* X (EXPN X (- N 1)))))
= EXPN
```

esempio:

```
? (EXPN 5 4)
= 625
```

2.7 La Traccia

La Traccia è un metodo che ci permette di osservare, passo dopo passo il calcolo di una funzione (spesso ricorsiva). Si usano le espressioni "mettere una funzione sotto traccia" per indicare che stiamo chiedendo al sistema di fornirci le informazioni date da questo operatore e "togliere la traccia di una funzione", quando vogliamo che ci venga fornito solo il risultato del computo. Le informazioni principali fornite dal computer sul processo di calcolo quando una funzione è sotto traccia sono le seguenti:

- per ogni chiamata indica il nome della funzione chiamata e i suoi argomenti
- per ogni risultato indica il nome della funzione e il risultato

Quanto abbiamo detto può essere reso più chiaro mediante un esempio. Mettiamo sotto traccia la funzione EXPN :

```
? (TRACE EXPN)
= (EXPN)

? (EXPN 5 4)
EXPN ---> X=5, Y=4
EXPN ---> X=5, Y=3
EXPN ---> X=5, Y=2
EXPN ---> X=5, Y=1
EXPN ---> X=5, Y=0
EXPN <--- 1
EXPN <--- 5
EXPN <--- 25
EXPN <--- 125
EXPN <--- 625
= 625
```

La freccia ---> indica una chiamata con gli argomenti,
la freccia <--- indica il ritorno della funzione con il risultato.
Nelle sequenza precedente, si vede che per calcolare (EXPN 5 4) abbiamo bisogno di aver già calcolato (EXPN 5 3) e così via.

Per togliere la traccia usiamo UNTRACE.

```
? (UNTRACE EXPN)
= (EXPN)
```

2.8 Altre funzioni

2.8.1 Fattoriale: FATT

```
? (DE FATT (N)
?   (IF (= N 0)
?     1
?     (* N (FATT (- N 1)))))
= FATT
```

```
? (FATT 5)
= 120
```

2.8.2 Numeri di Fibonacci: FIB

```
? (DE FIB (N)
?   (IF (< N 3)
?     1
?     (+ (FIB (- N 1)) (FIB (- N 2)))))
= FIB

? (FIB 10)
= 55
```

2.8.3 Massimo comun divisore (algoritmo di Euclide): MCD

```
? (DE MCD (X Y)
?   (IF (= Y 0)
?     X
?     (MCD Y (MODULO X Y))))
= MCD

? (MCD 91 65)
= 13
```

2.8.4 Rete di funzioni pari e dispari

Il predicato PARI è vero per ogni numero naturale pari e falso per ogni numero naturali dispari, analogamente il predicato DISPARI. Visti, come al solito, come funzioni che assumono solo i valori 1 e 0, avremo che la funzione PARI assume il valore 1 per ogni numero naturale pari e 0 per ogni naturale dispari. La funzione DISPARI assume il valore 1 per ogni numero naturale dispari e 0 per ogni naturale pari. Nella definizione di ciascuna di esse facciamo intervenire l'altra. Esse sono dette mutuamente ricorsive. Ecco la definizione in LISP:

```
? (DE PARI (N) (IF (= N 0) 1 (DISPARI (- N 1))))
= PARI

? (DE DISPARI (N) (IF (= N 0) 0 (PARI (- N 1))))
= DISPARI
```

? (PARI 4)
= 1

? (DISPARI 4)
= 0

È particolarmente interessante osservare queste due funzioni sotto traccia:

? (TRACE PARI DISPARI)
= (PARI DISPARI)

? (PARI 4)
PARI ---> N=4
DISPARI ---> N=3
PARI ---> N=2
DISPARI ---> N=1
PARI ---> N=0
PARI <--- 1
DISPARI <--- 1
PARI <--- 1
DISPARI <--- 1
PARI <--- 1
= 1

3. Liste, numeri di Gödel e codifica di formule

I cosiddetti numeri di Gödel sono un modo sofisticato per codificare funzioni (aritmetiche) mediante numeri interi, cioè mediante oggetti dell'aritmetica stessa. Questo permette di formulare enunciati sull' aritmetica nel formalismo dell' aritmetica. Ma un numero è molto meno rappresentabile di una formula per spiegare una funzione.

In LISP invece, le stesse formule usate per definire funzioni possono anche essere viste come oggetti dati, le liste, e trattate in modo molto formale. In questo capitolo presentiamo lo stretto indispensabile del mondo delle liste necessario per esprimere la funzione universale del capitolo 5.

3.1 Definizione semplificata

In LISP, le liste sono oggetti con parentesi. Questi oggetti sono costruiti

con la funzione LIST, per esempio:

? (LIST 2 5)
= (2 5)

Nella matematica, questo esempio, una coppia, è spesso utilizzato per definire l'insieme dei numeri razionali mediante una coppia di numeri interi. Non presenteremo qui la discussione completa, chi vuole può trovarla nei capitoli introduttivi di molti libri (si veda, ad esempio, il libro di H. Abelson e G. Sussman, "Structure and Interpretation of Computer Programs", MIT Press, 1985)

Per accedere agli elementi di una lista, ci sono due funzioni, CAR per accedere al primo elemento e CADR per il secondo; esempio:

? (CAR (LIST 2 5))
= 2

? (CADR (LIST 2 5))
= 5

Anche qui occorre ricordasi della polemica di quaranta anni fa: perché usare nomi così oscuri, criptografici, per esprimere delle operazioni così semplici? Non c'è altra riposta che la storia. I nomi traggono origine dalla prima machina (I.B.M.) nella quale è stato implementato il LISP: CAR significa "Contents of Address Register". Il significato di CADR verra spiegato successivamente.
Con questi tre funzioni LIST, CAR e CADR possiamo già costruire delle liste semplici ma non sono adatte per trattare strutture più complesse.

3.2 Definizione generale delle liste

Siamo adesso in grado di costruire liste con due elementi. Abbiamo bisogno di due altre funzioni: una che aggiunge un elemento a una lista e una che toglie un elemento della lista. La funzione che aggiunge si chiama CONS, è una funzione di costruzione (**cons**truct in inglese) a due argomenti: il primo è un oggetto LISP qualsiasi, il secondo deve essere una lista. Questa funzione costruisce una nuova lista combinando i due argomenti.

Esempio:

? (CONS 7 (LIST 3 5))
= (7 3 5)

Così partendo di una lista con due elementi (3 5) abbiamo ottenuto una lista
con tre elementi (7 3 5).
Per costruire una lista molto più lunga, possiamo iterare la costruzione con una funzione ricorsiva.

Esercizio: Definire una funzione che prende un numero come argomento e che
costruisce una lista con tutti i numeri dal argomento fino a 1 in ordine decrescente. Il nome della funzione sarà LL (per Lista Lunga).

Soluzione:
? (DE LL (N) (IF (= N 0) () (CONS N (LL (- N 1)))))
= LL

? (LL 19)
=(19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)

? (LL 0)
= ()

Osservazione: Si scrive () per la lista vuota.

Anche per le funzioni sulle liste, la traccia è molto interessante:

? (TRACE LL)
= LL

? (LL 3)
LL ---> N=3
LL ---> N=2
LL ---> N=1
LL ---> N=0
LL <--- ()
LL <--- (1)
LL <--- (2 1)
LL <--- (3 2 1)
= (3 2 1)

Mettendo in evidenza ciò che viene fatto passo dopo passo, si vede perché i numeri sono in ordine decrescente. Ovviamente dando un'altra definizione avremmo potuto ottenere una lista in ordine crescente.

Adesso siamo in grado di costruire liste di una lunghezza qualsiasi, però possiamo accedere solo al primo e al secondo elemento di una lista con le funzioni CAR e CADR del punto 3.1. Manca una funzione generale di accesso, corrispondente alla funzione CONS di costruzione generale. La funzione che (associata a CAR) permette di accedere a qualsiasi elemento della lista si chiama CDR. Questo nome è un acronimo di "Contents of Decrement Register" e deriva (come quello di CAR) dal nome assegnato nel primo computer I.B.M. che ha implementato il LISP.

La funzione CDR prende una lista come argomento, toglie il suo primo elemento e dà il resto della lista come risultato. Così, per accedere al terzo elemento di una lista, basta togliere il primo elemento e dopo accedere al secondo come nel seguente esempio:

```
? (DE TERZO (L) (CADR (CDR L)))
= TERZO

? (TERZO (CONS 7 (LIST 3 5)))
= 5
```

Una funzione di accesso all' n-esimo elemento di una lista si può definire in modo ricorsivo su N: basta togliere N-1 elementi e dopo accedere al primo. NESIMO sarà dunque una funzione che prende un numero N e una lista L come argomento.

Soluzione:

```
? (DE NESIMO (N L)
?   (IF (= N 1)
?     (CAR L)
?     (NESIMO (- N 1) (CDR L))))
= NESIMO

? (NESIMO 4 (LL 19))
= 15
```

3.3 La lista vuota

La lista vuota, scritta come (), è un oggetto molto importante nel mondo delle liste, un pò come il numero 0 nel mondo dell'aritmetica. Nelle definizioni per alternativa si trova spesso un caso particolare che ha a che fare con la lista vuota. È per questo motivo che esiste un predicato sulle liste per verificare se un oggetto è la lista vuota. Il predicato si chiama NULL: questo predicato è vero quando il suo argomento è la lista vuota, falso altrimenti. Proviamo per esempio a calcolare la lunghezza di una lista, cioè il numero di elementi in essa contenuti. Proviamo a dare una definizione informale di tipo ricorsivo della lunghezza di una lista. Possiamo dire che:

- la lunghezza della lista vuota è zero
- la lunghezza di una lista è uguale alla lunghezza della lista che si ottiene da essa
  privandola del suo primo elemento incrementata di 1.

Prima di esprimere questa definizione informale in LISP, osserviamo che si deve essere in grado di stabilire se una lista è vuota o no, dunque si deve usare il predicato NULL.

Soluzione in LISP:

```
? (DE LUNGHEZZA (L)
?   (IF (NULL L)
?     0
?     (+ 1 (LUNGHEZZA (CDR L)))))
= LUNGHEZZA

? (LUNGHEZZA (LIST 7 3 5))
= 3
```

Questa funzione è standard, cioè è predefinita in ogni sistema LISP col nome inglese LENGTH.

Naturalmente si può ricreare adesso tutto un calcolo nel mondo delle funzioni sulle liste in modo analogo a quanto fatto per le funzioni aritmetiche. È una estensione che si può trovare in un qualsiasi libro sul LISP. Qui vorrei solo presentare tre altri esempi:

3.4 Il prodotto scalare

Una lista può essere vista come la rappresentazione di un vettore. Il prodotto scalare, PS, è una funziono che prende due vettori come argomento e dà un numero come risultato. Esempio:

```
? (DE PS (X Y)
?   (IF (NULL X)
?     0
?     (+ (* (CAR X) (CAR Y)) (PS (CDR X) (CDR Y)))))
= PS

? (PS (LIST 3 7 5) (LIST 3 -2 2))
= 5
```

3.5 Aggiungere una lista a un'altra lista

```
? (DE AGGIUNGERE (X Y)
?   (IF (NULL X)
?     Y
?     (CONS (CAR X) (AGGIUNGERE (CDR X) Y))))
= AGGIUNGERE

? (AGGIUNGERE (LIST 3 5 7) (LIST 3 -2 2))
= (3 5 7 3 -2 2)
```

Questa funzione è standard in ogni sistema LISP col nome inglese APPEND.

3.6 Sostituzione

I tre esempi precedenti sono tutti definizioni per alternativa con due casi ed usano tutti il predicato NULL. Per la sostituzione abbiamo bisogno di tre casi e due predicati. È una funzione che sostituisce in una lista un numero con un altro. La chiamiamo SOST, prende tre argomenti: il vecchio numero, il nuovo numero e la lista. Costruisce una nuova lista che contiene dappertutto il nuovo numero al posto del vecchio:

```
? (DE SOST (X Y L)
?   (IF (NULL L)
?     ()
?     (IF (= X (CAR L))
?       (CONS Y (SOST X Y (CDR L)))
?       (CONS (CAR L) Y (SOST X Y (CDR L))))))
= SOST
```

3.7 Liste di liste

Fin qua, abbiamo sempre costruito liste che contengono numeri come elementi, queste liste sono dette "piatte". Era un modo per familiarizzarsi con questo mondo utilizzando oggetti conosciuti da tutti. Ma una lista LISP può contenere altri oggetti, particolarmente altre liste. Questo permette di costruire oggetti di una complessità qualsiasi.

Esempio: una matrice a due dimensioni può essere rappresentata come una lista che contiene due sottoliste:

```
? (LIST (LIST 3 4) (LIST 2 -5))
= ((3 4) (2 -5))
```

Per accedere agli elementi basta combinare delle operazioni successive di CAR e CDR .
Esempio: calcolare il determinante di una matrice a due dimensioni:

```
? (DE DET (M)
?   (-
?     (* (CAR (CAR M)) (CADR (CADR M)))
?     (* (CAR (CADR M)) (CADR (CAR M)))))
= DET

? (DET (LIST (LIST 3 4) (LIST 2 -5)))
= -23
```

Non è molto comodo trattare le matrici in questo modo e neppure facile da leggere. Quando si ha a che fare solo con due dimensioni non ne vale proprio la pena. L'interesse sta nel fatto che il processo può essere generalizzato ad un numero qualsiasi di dimensioni.

3.8 Profondità

Nel paragrafo 3.2 abbiamo costruito liste con una lunghezza qualsiasi, mettendo sempre elementi nella prima posizione di una lista già esistente. Si può studiare un processo simile agendo sul livello delle parentesi di una lista. Possiamo creare liste molto profonde continuando a inserire parentesi attorno a una lista già esistente. Ad esempio, l'operatore LP (lista profonda) costruisce una lista con N livelli di parentesi:

```
? (DE LP (N)
?   (IF (= N 0)
?     ()
?     (CONS (LP (- N 1)) ())))
= LP

? (LP 5)
=(((((())))))
```

Si osservi che la formula ricorsiva si trova questa volta al primo argomento della funzione CONS.

Quando analizziamo una lista con struttura qualsiasi dobbiamo affrontare due problemi collegati al fatto che:

- non conosciamo la lunghezza delle liste incontrate
- non sappiamo se gli elementi sono a loro volta altre liste

Per rispondere alla prima domanda abbiamo già gli strumenti da utilizzare: mediante il predicato NULL si può calcolare la lunghezza di una lista qualsiasi, oppure - più in generale – possiamo sapere quando abbiamo esaminato una lista fino alla fine. Per affrontare il secondo problema ci serve un altro predicato.

3.9 Il predicato ATOM

Un oggetto LISP si dice atomico quando non è una lista, cioè quando le operazioni CAR e CDR sono proibite. Abbiamo incontrato per adesso solo numeri come oggetti atomici. Il predicato ATOM è vero per i numeri e falso per le liste. Questo permette di distinguere tra liste ed altre cose, e, in realtà, ci permette di analizzare liste di un qualsiasi livello di complessità. Esempio: appiattire una lista, cioè fare la proiezione sul

piano, oppure raccogliere tutti gli elementi in una lista piatta.

```
? (DE PIATTA (L)
?   (IF (NULL L)
?     ()
?     (IF (ATOM L)
?       (LIST L)
?       (APPEND (PIATTA (CAR L)) (PIATTA (CDR L))))))
= PIATTA

? (PIATTA (LIST (LIST 3 4) (LIST 2 -5)))
= (3 4 2 -5)
```

Osservazioni:

- È una definizione con tre casi:
    - l'argomento è la lista vuota: non c'è niente da fare;
    - l'argomento non è una lista, cioè è un atomo: lo mettiamo in lista con la funzione LIST;
    - in tutti gli altri casi ci sono chiamate ricorsive per appiattire il primo elemento e il resto della lista, ed aggiungere i due risultati con APPEND.

- Usa i due predicati NULL ed ATOM.

- Contiene due chiamate ricorsive sulle parti CAR e CDR dell'argomento.

Questi tre punti sono caratteristici per ogni funzione che lavora sulle liste di liste. Studiamo un secondo esempio:

3.10 Sostituzione in una lista di liste.

```
? (DE SOST (X Y L)
?   (IF (NULL L)
?     ()
?     (IF (ATOM L)
?       (IF (= X L) Y L)
?       (CONS (SOST X Y (CAR L)) (SOST X Y (CDR L))))))
```

= SOST

Lo scopo di questo capitolo è stato quello di mostrare come si possano codificare funzioni aritmetiche in LISP sotto forma di liste. Siamo adesso in grado di costruire formule della stessa struttura e complessità delle nostre definizioni del capitolo 2. Però queste liste contengono solo numeri. Dobbiamo introdurre ancora variabili e nomi di operatori e di funzioni, cioè dati simbolici.
Questo lo faremo nella seconda parte di quest'articolo nella quale introdurremo anche altre cose e scopriremo in modo del tutto naturale che non possiamo costruire funzioni per rispondere a qualsiasi domanda ci poniamo.

4. Simboli, funzioni e funzionali

Il modo in cui una sequenza di cifre è vista dall' uomo o dal computer spesso non è ambiguo: tutti la vedono come numero. Se invece ci troviamo di fronte a una sequenza di lettere o una sequenza mista di lettere e di cifre, la situazione è diversa. Può darsi che si tratti di una variabile (come, ad esempio, nel caso di X), di una funzione (F), di una espressione con un indice (A_1) o di una parola italiana (AGGIUNGERE) che è nello stesso tempo il nome di una funzione. Per non preoccuparsi di alcune ambiguità di questo tipo, esiste nel LISP un construtto chiamato "apostrofo".

4.1 Costruzione di simboli

L'oggetto che segue un apostrofo sarà considerato come dato simbolico, ad esempio:

? 'GATTO
= GATTO

Questo costrutto permette anche di introdurre delle parole nelle nostre strutture dati. Possiamo, ad esempio, costruire delle frasi in italiano:

? (LIST 'IL 'GATTO 'MANGIA 'IL 'TOPO)
= (IL GATTO MANGIA IL TOPO)

oppure delle formule matematiche

? (LIST 'COS (LIST 'X '* 'X))
= (COS (X * X))

Quando una lista da costruire contiene solo dati simbolici, si può mettere un solo apostrofo davanti alla lista, cioè dare la lista intera come costante e risparmiare la chiamata alla funzione LIST, ad esempio:

? '(IL GATTO MANGIA IL TOPO)
= (IL GATTO MANGIA IL TOPO)

? '(* 5 (+ 34 78))
= (* 5 (+ 34 78))

Successivamente, questi costrutti possono essere manipolati e analizzati come si vuole, ad esempio, consideriamo la trasformazione della notazione infissa in quella prefissa e vice versa nel caso di operatori binari:

```
? (DE TRASF (X)
?   (IF (ATOM X)
?     X
?     (LIST
?       (TRASF (CADR X))
?       (TRASF (CAR X))
?       (TRASF (TERZO X)))))
= TRASF

? (TRASF '(* 5 (+ 34 78)))
= (5 * (34 * 78))

? (TRASF '(5 * (34 * 78)))
= (* 5 (+ 34 78))
```

L'esempio precedente è semplicemente una trasformazione dell'ordine degli elementi atomici della lista: ovunque il primo elemento è scambiato col secondo. Nel corso di questo riordinamento basta analizzare la struttura della lista con il predicato ATOM, non c'è bisogno di riconoscere gli elementi.

4.2 Funzioni

In modo molto semplificato possiamo rappresentare le funzioni come simboli conosciuti dal sistema, cioè come simboli utilizzabili nelle formule di calcolo. Ad esempio: +, -, *, /, QUOTIENT, MODULO, QUAD, LIST, CONS .

Ma un simbolo che sappiamo che è nome di funzione, cioè che possiede una definizione funzionale, può anche fare qualcosa dippiù: può essere applicato a un argomento o a una lista di argomenti.
Ci sono due funzioni standard, FUNCALL ed APPLY, che realizzano queste applicazioni. In un certo senso sono il contrario dell' apostrofo: invece di considerare un oggetto come dato simbolico, lo considerano come dato funzionale e , così facendo, fanno innescare un calcolo. FUNCALL ed APPLY sono anche chiamate "funzionali" perché hanno una funzione come primo argomento.

Esempio:

? (FUNCALL 'QUAD 5)
= 25
? (FUNCALL 'CAR (LIST 3 7))
= 3
? (FUNCALL 'CDR (LIST 3 7))
= (7)

? (APPLY 'QUAD (LIST 5))
= 25
? (APPLY '* (LIST 3 7))
= 21

La differenza tra FUNCALL ed APPLY risiede solo nel loro secondo argomento che per FUNCALL è l'argomento della funzione da applicare, mentre per APPLY è la lista di tutti gli argomenti sui quali effettuare il calcolo. Il primo argomento sia di FUNCALL sia di APPLY è sempre un simbolo con una definizione di funzione valida. Negli esempi sopra riportati, questo simbolo era sempre dato come costante, ed era introdotto proprio mediante l'apostrofo. Ma l'interesse di questi funzionali, naturalmente, proviene dalle possibilità di ottenere proprio il simbolo come risultato di un calcolo.

Come esempio considereremo quello del valutatore aritmetico. Si vuole scrivere una funzione VAL che prenda come argomento una espressione

aritmetica sotto forma di lista e che produca come risultato un numero che è proprio il valore di questa espressione. Cioè dobbiamo simulare le regole del calcolo che permettono di effettuare correttamente i vari passaggi.

In primo luogo osserviamo che un' espressione aritmetica sarà rappresentata nella notazione polacca prefissa sotto forma di una lista, come abbiamo già visto nel capitolo 2:

```
(OPERATORE  OPERANDO1  OPERANDO2)
```

L'operatore sarà uno dei simboli "+", "-", "*" e "/". Per estrarre questo simbolo useremo la funzione CAR, mentre si può accedere al posto occupato da OPERANDO1 ed OPERANDO2 mediante le funzioni CADR e TERZO. Possono esserci altre espressioni aritmetiche, che devono essere semplificate prima di potere applicare l'operazione corrispondente all' operatore. Il processo è ricorsivo. Ha la stessa struttura della funzione TRASF, ma invece di avere solo uno spostamento e una costruzione di lista qui si ha un'applicazione con FUNCALL per un calcolo aritmetico:

```
? (DE VAL (X)
?   (IF (ATOM X)
?     X
?     (FUNCALL (CAR X) (VAL (CADR X)) (VAL (TERZO X)))))
= VAL

? (VAL '(* 5 (+ 34 78)))
= 560
```

Per coloro che preferiscono la notazione infissa si ha:

```
? (VAL (TRASF '(5 * (34 * 78))))
= 560
```

Abbiamo incontrato adesso tante funzioni con la stessa struttura o con strutture poco differenti che vale la pena di formalizzare un poco questo punto.

4.3 Schemi di programmi e funzionali

Osserviamo ancora due funzioni molto simili. Vogliamo costruire prima una funzione, LQUAD, che prende una lista di numeri come argomento e

che produce una lista dei suoi quadrati come risultato. Per esempio:

```
? (DE LQUAD (L)
?   (IF (NULL L)
?     ()
?     (CONS (QUAD (CAR L)) (LQUAD (CDR L)))))
= LQUAD
? (LQUAD (LIST 3 7 -25 0))
= (9 49 625 0)
```

In secondo luogo, vogliamo costruire una funzione, LFATT, che ha la stessa struttura ma che, invece del quadrato, calcoli i fattoriali di tutti gli elementi della lista, ad esempio:

```
? (DE LFATT (L)
?  (IF (NULL L)
?    ()
?    (CONS (FATT (CAR L)) (LFATT (CDR L)))))
= LFATT
? (LFATT (LIST 3 7 5 0))
= (6 5040 120 1)
```

Vedendo definizioni come queste, a un matematico sorge, in modo naturale, il desiderio di unificarle e generalizzarle in modo da poterle poi usare per funzioni qualsiasi e non solo per il quadrato e il fattoriale. Il nome della funzione deve essere dato come argomento. La soluzione a questo problema è data dalla funzione SUPERF, usando FUNCALL:

```
? (DE SUPERF (F L)
?   (IF (NULL L)
?     ()
?     (CONS (FUNCALL F (CAR L)) (SUPERF F (CDR L))))
= SUPERF

? (SUPERF 'QUAD (LIST 3 7 -25 0))
= (9 49 625 0)

? (SUPERF 'FATT (LIST 3 7 5 0))
= (6 5040 120 1)

? (SUPERF CUBO (LIST 3 7 -25 0))
= (27 343 15625 0)
```

SUPERF è dunque un funzionale scritto da noi, ma è anche una funzione standard in ogni sistema LISP col nome inglese di MAPCAR.

Lo scopo dell'introduzione di simboli in questo discorso era quello di avere strumenti per rappresentare e manipolare funzioni. Le operazione di base che abbiamo incontrato sono:

- la creazione del simbolo mediante l'apostrofo e
- l'applicazione con FUNCALL o APPLY

Abbiamo studiato, infatti, soprattutto simboli con una definizione funzionale. Considereremo adesso predicati più generali.

4.4 Uguaglianza di simboli

Per confrontare numeri abbiamo usato il segno solito della matematica, =. Esiste anche un altro predicato, EQ, che è in grado di effettuare il confronto anche tra simboli. Una formula con EQ è vera quando i suoi argomenti sono lo stesso simbolo. Come esempio considereremo un semplice sistema di traduzione da una lingua a un'altra: Vogliamo scrivere una funzione che prenda una parola italiana come argomento e produca la parola inglese corrispondente come risultato, cioè vogliamo ottenere una specie di dizionario elettronico. Ogni parola sarà rappresentata da un simbolo.

```
? (DE TRADUC (X)
?   (IF (EQ X 'IL) 'THE
?     (IF (EQ X 'GATTO) 'CAT
?       (IF (EQ X 'MANGIA) 'EATS
?         (IF (EQ X 'TOPO) 'MOUSE X)))))
= TRADUC
? (TRADUC 'TOPO)
= MOUSE
```

C'è un modo più breve (che fa uso di COND, CASE e SELECTQ) per esprimere questa catena di IF, ma noi non abbiamo lo spazio per discutere in dettaglio questo aspetto. Anche il vocabolario del traduttore è ridotto a quattro parole. È chiaro che non si vuole mostrare un traduttore reale, ma solo una manipolazione di simboli che non sono funzioni.

Un altro esempio è fornito dalla sostituzione di un simbolo con un altro. Nel punto 3.10 abbiamo esaminato il problema della sostituzione di un elemento con un altro in una lista di numeri usando il predicato = . Questa definizione può essere generalizzata a simboli e numeri usando il predicato EQ al posto di = . Ecco la definizione modificata:

```
? (DE SOST (X Y L)
?   (IF (NULL L)
?     ()
?     (IF (ATOM L)
?       (IF (EQ X L) Y L)
?       (CONS (SOST X Y (CAR L)) (SOST X Y (CDR L))))))
= SOST
```

Possiamo classificare i simboli come segue:

- simboli che non hanno definizione funzionale: GATTO, IL, TOPO
- simboli che rappresentano funzioni scritte da noi: QUAD, FATT, TRADUC
- simboli che rappresentano funzioni standard: +, -, *, LIST, CAR, FUNCALL

Per distinguere tra loro esiste un altro predicato che adesso andremo a considerare.

4.5 Il predicato GETDEF

La funzione GETDEF prende un simbolo come argomento e produce una lista come risultato. Mediante questa lista si può individuare il tipo di simbolo e di funzione:

- quando la lista è vuota: non c'è definizione funzionale
- quando il primo elemento è il simbolo "DE": si tratta di una funzione scritta da noi che è chiamata anche "funzione utente". La lista è la definizione che abbiamo usato al momento in cui abbiamo introdotto la funzione.
- altrimenti: si ha una funzione standard e la lista è un indirizzo in linguaggio macchina.

Esempi:

? (GETDEF 'GATTO)
= ()

? (GETDEF 'QUAD)
= (DE QUAD (X) (* X X))

? (GETDEF 'CONS)
= (DS CONS SUBR2 30567)

Adesso abbiamo raccolto tutti gli strumenti che occorrono per definire una funzione aritmetica in LISP, per codificare la definizione sotto la forma "(DE.....)" e per ritrovarla ed analizzarla sotto forma di lista.

4.6 Il processo dell' applicazione di una funzione utente

La prima funzione utente incontrata è stata il quadrato QUAD, vediamo che cosa succede quando si deve calcolare una formula (FUNCALL 'QUAD 5) :

- QUAD e` riconosciuto come funzione utente, la lista della sua definizione è
  (DE QUAD (X) (* X X))

- Il parametro è X, l'espressione definienda è (* X X)

- Sostituiamo in modo uniforme l'argomento 5 al posto di X nell'espressione definienda:

  (SOST 'X 5 '(* X X)) ---> (* 5 5)

- Valutiamo il risultato come un' espressione aritmetica.

La quarta funzione utente incontrata è stata l'elevazione a potenza, EXPN, vediamo che cosa succede quando si deve calcolare una formula del tipo (FUNCALL 'EXPN 5 4):

- EXPN è riconosciuta come funzione utente, la lista della sua definizione è

(DE EXPN (X N) (IF (= N 0) 1 (* X (EXPN X (- N 1)))))

- I parametri sono X e N, l'espressione definienda è

(IF (= N 0) 1 (* X (EXPN X (- N 1))))

- Sostituiamo in modo uniforme l'argomento 5 a X e l'argomento 4 a N nell'espressione definienda:

(IF (= 4 0) 1 (* 5 (EXPN 5 (- 4 1))))

- Valutiamo il risultato come un'espressione condizionale.

A un certo punto il processo chiede di calcolare la formula (* 5 (EXPN 5 3)) e così di seguito fino ad arrivare a un'espressione (* 5 (* 4 (* 3 (* 2 1)))) che contiene solo numeri e può essere calcolata usando solo funzioni standard. Ci sono varie regole di riduzione e di ordine di riduzione rispetto all'espressione condizionale e al calcolo degli argomenti di funzioni (Call by name, call by value, etc). Queste non sono legate alla sintassi particolare del LISP. Possono essere studiate in dettaglio indipendentemente dal LISP in qualsiasi libro sulla calcolabilità.

5. Il valutatore come funzione universale

In questo capitolo presenteremo una strategia e un insieme di cinque nuove funzioni che permettono di eseguire la serie di passaggi discussa sopra sotto forma di manipolazione di liste. Ognuna di esse richiama almeno una delle altre e tutte insieme rappresentano la funzione universale: una funzione che ha come argomento un'altra funzione e i suoi argomenti e che calcola il suo risultato, cioè una funzione FUNCALL e una funzione APPLY scritte da noi, come funzioni utente.

Il fatto di manipolare liste simboliche, invece di numeri di Gödel, rende il processo più trasparente.

Questo processo è diviso in due parti: l'analisi delle espressioni e l'analisi delle applicazioni funzionali.

Una espressione può essere:
- un numero
- una espressione condizionale con un IF

- un'altra formula in notazione polacca prefissa che sarà vista come applicazione di una funzione.

In un'applicazione, la funzione può essere:
- sconosciuta, in questo caso siamo di fronte a un errore, a una formula sbagliata
- una funzione standard
- una funzione utente.

Queste sei regole si codificano mediante due funzioni, ciascuna delle quali richiede l'analisi di tre casi. Dobbiamo sapere come riconoscere i casi, cioè sapere quale predicato LISP usare, e come agire secondo le regole.

5.1 Codice del valutatore

La funzione VALUTA corrisponde all'analisi dell'espressione, il predicato ATOM serve nel caso di un numero, il confronto mediante il simbolo IF per una espressione condizionale:

```
? (DE VALUTA (X)
?  (IF (ATOM X)
?    X
?    (IF (EQ 'IF (CAR X))
?     (VALUTA-IF (CDR X))
?     (APPLICA (GETDEF (CAR X)) (VALUTA-LIST (CDR X))))))
= VALUTA
```

Nel caso di un numero non bisogna fare nulla il numero vale se stesso. Nel caso di un'espressione condizionale, viene usata un'altra funzione, VALUTA-IF, la quale prende la decisione:

```
? (DE VALUTA-IF (L)
?   (IF (VALUTA (CAR L))
?     (VALUTA (CADR L))
?     (VALUTA (TERZO L))))
= VALUTA-IF
```

Il suo argomento è composto dal predicato, dalla parte vero e dalla parte falso, sotto forma di una lista con tre elementi. Dobbiamo in primo luogo valutare il predicato e successivamente, a seconda del suo valore, la parte

corrispondente a vero o quella corrispondente a falso. Si vede che solo una delle due parti viene valutata. Qui si trova la prima chiamata ricorsiva al valutatore.

Nel caso delle applicazioni, sono usate altre due funzioni ausiliarie, APPLICA e VALUTA-LIST. APPLICA è la funzione universale ancora da definire. VALUTA-LIST è una funzione molto semplice, prende come argomento una lista di espressioni e produce come risultato una lista di numeri ottenuti successivamente con chiamate ricorsive alla funzione VALUTA. Qui dentro si trova la seconda chiamata ricorsiva al valutatore:

```
? (DE VALUTA-LIST (L) (MAPCAR 'VALUTA L))
= VALUTA-LIST
```

Adesso sono pronti i due argomenti della funzione APPLICA: Il primo argomento è una definizione di una funzione sotto forma di lista, come data da GETDEF. Il secondo argomento è una lista di argomenti per questa funzione. I predicati usati per distinguere i tre casi sono: NULL, nel caso di errore e il confronto con la parole chiave DS, per una funzione standard:

```
? (DE APPLICA (F L)
?   (IF (NULL F)
?     (ERROR 'APPLICA 'FUNZIONE 'SCONOSCIUTA)
?     (IF (EQ 'DS (CAR F))
?       (APPLY (CADR F) L)
?       (VALUTA (LSOST (TERZO F) L (TERZO (CDR F)))))))
= APPLICA
```

Nel caso di errore, non si deve fare più niente tranne che chiamare la funzione ERROR del sistema per abbandonare il processo.
Nel caso di una funzione standard, siamo arrivati a una funzione primitiva aritmetica e possiamo ottenere il suo valore mediante APPLY.
L'ultimo caso - il più interessante - è quello di una funzione utente che mette in opera il processo di sostituzione.

Ci troviamo dunque messi a confronto con

- una lista di parametri (che proviene da GETDEF)
- una lista di argomenti (che proviene da VALUTA-LIST)
- una espressione definienda (che proviene da GETDEF)

Non c'è una sola sostituzione ma tante quanti sono i parametri. Per ogni parametro chiamiamo la nostra funzione SOST definita prima. La funzione LSOST ha tre argomenti:

- una lista di simboli
- una lista di valori
- una lista qualsiasi che contiene normalmente i simboli da sostituire

ed essa applica la funzione SOST a ogni elemento della lista di simboli.

```
? (DE LSOST (X Y L)
?   (IF (NULL X)
?     L
?     (LSOST (CDR X) (CDR Y) (SOST (CAR X) (CAR Y) L))))
= LSOST
```

Dopo il processo di sostituzione, ci troviamo con un'espressione che non contiene più variabili. In alcuni casi può essere già semplificata come espressione aritmetica semplice. A questo punto arriva la terza chiamata ricorsiva alla funzione VALUTA.

5.2 Uso del valutatore

Per esaminare passo passo il processo di riduzione, conviene mettere il valutatore (così come altre funzioni) sotto traccia, ad esempio:

```
? (TRACE VALUTA APPLICA)
= (VALUTA APPLICA)
? (VALUTA '(QUAD 5))
```

VALUTA ---> X=(QUAD 5)
VALUTA ---> X=5
VALUTA <--- X=5
APPLICA ---> F=(DE QUAD (X) (* X X)) L=(5)
VALUTA ---> X=(* 5 5)
VALUTA ---> X=5
VALUTA <--- 5
VALUTA ---> X=5
VALUTA <--- 5
VALUTA <--- 25
APPLICA <--- 25
VALUTA <--- 25
= 25

Altri esempi con funzioni ricorsive come EXPN sono più interessanti ma troppo lunghi per essere presentati qui.

5.3 Il problema della fermata

Ricordiamo la definizione del fattoriale FATT; mettiamo tale funzione sotto traccia e proviamo a calcolare (FATT -1). Che cosa otteniamo?

? (TRACE FATT)
= (FATT)
? (FATT -1)
FATT ---> N=-1
FATT ---> N=-2
FATT ---> N=-3
...

Questo processo non si ferma mai. Ricordiamo che nella definizione del fattoriale c'è il confronto con lo zero; ma se il valore dato all'ingresso è negativo non potremo mai arrivare allo zero. La funzione FATT è definita solo per numeri naturali. Questo osservazione ci induce a considerare il problema seguente: È possibile scrivere una funzione LISP, FERMATA, che prenda come argomento un'altra funzione LISP e i suoi argomenti e invece di calcolare subito il risultato prima mi dice se tale risultato esiste o no, ad esempio:

? (FERMATA 'FATT -1)
= FALSO

```
? (FERMATA 'FATT 5)
= VERO
```

La teoria della calcolabilità, purtroppo, ci dice che, nel caso generale, è impossibile scrivere una tale funzione.

5.4 Il problema dell' equivalenza

Consideriamo adesso la definizione seguente:

```
? (DE MISTERO (N R)
?   (IF (= N 0)
?     R
?     (MISTERO (- N 1) (* N R))))
= MISTERO
```

Che cosa calcola questa funzione? Si può dire che per ogni N (numero naturale):

```
(MISTERO N 1) = (FATT N)
```

Cioè possiamo usare la funzione MISTERO per definire in un altro modo il fattoriale :

```
? (DE FATT2 (N) (MISTERO N 1))
= FATT2

? (FATT2 5)
= 120
```

FATT e FATT2 sono equivalenti, hanno due definizioni diverse, ma calcolano la stessa funzione aritmetica.

Esaminiamo adesso il problema seguente: è possibile scrivere una funzione LISP, EQUIV, che prenda come argomento due simboli e analizzi le loro definizioni per decidere se esse calcolano o no la stessa funzione? Ad esempio:

```
? (EQUIV 'FATT 'FATT2)
= VERO
```

? (EQUIV 'FATT 'QUAD)
= FALSO

Anche in questo caso, purtroppo, la teoria della calcolabilità ci dice che nel caso generale, è impossibile scrivere una tale funzione.

Questi ultimi due esempi mostrano che nonostante la potenza della formalizzazione del calcolo, vi sono delle cose che il computer non può fare. Ma questa formalizzazione è almeno un mezzo per vedere chiaro.

5.5 La torre dei valutatori

Abbiamo sfruttato finora il fatto che una formula LISP è rappresentata come lista. In questo modo è stato possibile scrivere la funzione VALUTA che realizza i vari passi di semplificazione mediante operazioni sulle liste. Ad esempio, consideriamo la formula LISP: (QUAD 5), il risultato di questo esempio è 25. In questo esempio la chiamata al valutatore è data da (VALUTA '(QUAD 5)), ma poiché anche quest'ultima è una formula LISP essa può essere data come ingresso al valutatore e avremmo: (VALUTA '(VALUTA '(QUAD 5))).

Sorge a questo punto la domanda: ma il valutatore può valutare se stesso? La risposta è: in linea di principio SI; ma perché questo possa effettivamente avvenire sono necessarie le due cose seguenti che per semplicità - volendo scrivere un valutatore minimo - non sono contenute nel valutatore che abbiamo presentato prima:

- si deve sapere come è trattato l'apostrofo
- si deve sapere come fare, o evitare, la sostituzione quando si è in presenza di dati simbolici.

Quindi a differenza del problema della fermata o di quello dell'uguaglianza tra funzioni LISP che non possono essere decisi da altre funzioni LISP, si possono scrivere valutatori LISP che valutano se stessi (anzi, questo è un lavoro che è sempre stato fatto fin da quando il LISP è nato). Questi sono utilizzati, all'inizio, in quella che viene chiamata fase di "bootstrap" per creare nuovi sistemi LISP o per trasferire un sistema su un diverso computer.