LISP

Teoria dei Grafi

1988–89

1 Liste

1. **Atomi** sono numeri o parole.
2. **Liste** sono
   (a) la lista vuota NIL o ()
   (b) se $t_1, \ldots, t_n$ sono atomi o liste, $(t_1 \ldots t_n)$.
3. **Operazioni sulle liste** sono
   - **car** (primo elemento della lista), per 'Content Address part of Register'
   - **cdr** (lista senza il primo elemento), per 'Content Decrement part of Register'
   - **last** (lista consistente solo dell'ultimo elemento)
   - **length** (lunghezza della lista)
   - **cons** (dati a e la lista x, la lista x con a aggiunto come primo elemento)
   - **append** (concatenazione di liste)
   - **list** (costruttore di liste)
   - **reverse** (invertitore del primo livello di lista).

Le liste sono implementate come **dotted pairs**, cioé come alberi binari, ciascun nodo dei quali fornisce un elemento della lista, ed un puntatore al resto della lista. Questo spiega perché car e cdr sono usati come fondamentali (danno i due costituenti dei nodi dell'albero). L'operazione inversa é cons, poiché

$$(\text{cons } (\text{car } x)\ (\text{cdr } x)) = x\ .$$

Le implementazioni di car e cdr introducono solo nuovi puntatori. Cons richiede l'introduzione di una nuova coppia puntata, con due puntatori al nuovo atomo e alla vecchia lista (questo rappresenta la nuova lista).

Append copia la prima lista e poi aggiunge un puntatore dalla fine di essa alla seconda lista. Se non copiasse, modificherebbe la prima lista (anche questa versione é implementata: **nconc**).

Due oggetti possono essere uguali perché puntano allo stesso oggetto (uguaglianza estensionale, **equal**), o perché sono lo stesso oggetto (uguaglianza intensionale, **eq**). Non c'é differenza fra le due uguaglianze solo quando ci si riferisce ad atomi.

C'é la possibilitá di effettuare interventi chirurgici sulle liste: nconc é uno, **rplaca** e **rplacd** sono altri (che, rispettivamente, rimpiazzano i puntatori sinistro o destro, cioé corispondenti a car e cdr, del primo argomento con il secondo argomento). Tali operazioni sono pericolose, perché muovere i puntatori significa modificare anche i valori di tutte le funzioni che in precedenza puntavano su di essi, e gli effetti sono difficilmente controllabili.

2 Funzioni e macro

Le funzioni possono essere definite in tre modi:

1. (**defun** $f\ (x_1 \ldots x_n)\ a$)
   La funzione $\lambda x_1 \ldots x_n.\, a$ riceve il nome f.

2. (**lambda** $(x_1 \ldots x_n)\ a$)
   Usata per usare la funzione $\lambda x_1 \ldots x_n.\, a$ senza darle un nome specifico.

3. (**def** f (lambda $x_1 \ldots x_n\ a$))
   Ha lo stesso effetto della prima, cioé pone f uguale alla funzione $\lambda x_1 \ldots x_n.\, a$.

I parametri occorrenti nella definizione (le x_i) sono vincolati. Se hanno un valore prima della chiamata di f, tale valore é registrato, e ripristinato alla fine della chiamata.

Gli altri eventuali parametri occorrenti in a sono liberi, ed i loro valori posono essere modificati durante l'esecuzione.

Le **macro** sono definite esattamente come le funzioni, usando **defmacro** al posto di defun. Esse sono peró trattate diversamente, perché non vengono eseguite, ma tradotte (ed una sola volta in una stessa compilazione). Si esegue poi la funzione risultante dalla loro traduzione.

Parametri opzionali si devono far precedere da &**optional** se singoli, e &**rest** se una sequenza.

3 Valutazione

Una lista $(f\ a_1 \ldots a_n)$ si valuta per **call by value** induttivamente, valutando prima tutti i termini che compaiono in essa, e poi considerando il primo termine come una funzione, ed applicandolo ai rimanenti termini, considerati come variabili.

I numeri (e le liste speciali nil e t) sono valutati come sé stessi, ed i rimanenti simboli vengono valutati in base alle loro assegnazioni. Ci sono peró alcune eccezioni: non si valutano

1. i nomi di procedura (per evitare di doverli far precedere da quote ogni volta)

2. il primo argomento di **setq**, (per 'set quote', cioé assegnazione) (perché scopo dell'assegnazione é quello di assegnare un valore a un simbolo che non l'ha ancora, e che non puó quindi essere valutato)

3. simboli preceduti da ' (o secondi argomenti di liste il cui primo argomento é **quote**). Le funzioni vengono quotate con il simbolo speciale $\#'$.[1]

[1] Il meccanismo di quote é essenziale per permettere il doppio livello di senso e significato. In Lisp il significato é preponderante, perché tutto é valutato, e quote permette il riferimento al senso. In ricorsivitá, il senso é preponderante, perché i numeri sono visti come tali, e per vederli come codifiche di programmi si deve usare un unquote, cioé il Teorema di Enumerazione.

4. elementi di liste precedute da ‘ (**backquote**), a meno che non siano preceduti da una virgola (questo é utile nelle definizioni di macro, in cui i parametri non vengono valutati)

5. macro, che sono prima tradotte e poi valutate.

Ci sono anche alcune procedure speciali di valutazione:

1. **eval** porta la valutazione a compimento, invece di fare solo un passo di essa. Ad esempio, se b é stato assegnato ad a, e c a b, chiamare a al prompt produce b, ma chiamare (eval a) produce c. Nota che eval é proprio il meccanismo di valutazione usato da Lisp.

2. **funcall** e **apply** differiscono solo per come sono presentati i parametri, e applicano una funzione agli argomenti. Cioé sia (funcall f $a_1 \ldots a_n$) che (apply f ($a_1 \ldots a_n$)) producono $f(a_1, \ldots, a_n)$. Nota che f é qui una variabile, che non deve essere stata necessariamente definita, e quindi l'ultima espressione darebbe errore in generale. In altre parole, funcall e apply sono funzionali, ed f é una variabile funzionale.

3. **mapcar** produce una valutazione parallela di f sugli argomenti indicati nella lista: (mapcar f ($a_1 \ldots a_n$)) produce ($f(a_1) \ldots f(a_n)$).

4 Assegnazioni e vincoli

Le assegnazioni possono essere di tre tipi:

1. **globali** (a variabili)
   L'assegnazione generale é (**setq** x y), che assegna il valore di y ad x (e corrisponde a ‘$x := y$’). Ci sono peró altre assegnazioni specifiche, che corrispondono a particolari funzioni.

   Ad esempio, (**pop** x) modifica il valore di x in (cdr x), cioé toglie ad una lista il primo elemento (e corrisponde a ‘$x :=$ (cdr x)’). E (**push** a x) modifica il valore di x in (cons a x), cioé aggiunge ad una lista x l'elemento a in testa (e corrisponde a ‘$x :=$ (cons a x)’). La differenza fra queste due assegnazioni e le funzioni cdr e cons é che queste ultime restituiscono gli stessi valori delle prime (al prompt), ma non modificano il valore corrente di x.

2. **locali** (a parametri in procedure)
   (**let** $((x_1\ a_1) \ldots (x_n\ a_n))\ t$) restituisce t in cui le variabili x_i sono state istanziate con i valori delle a_i. Il let assegna dunque parametri e, se t é il corpo di una funzione con parametri x_i, ha lo stesso effetto di calcolare la funzione sugli argomenti a_i.

   L'assegnazione dei parametri é effettuata in parallelo. Per assegnare i parametri in serie, si usa **let*** al posto di let.

3. **proprietá**
   Le proprietá sono considerate come data bases ordinati. Ci sono tre istruzioni, che permettono rispettivamente di aggiungere, togliere e cercare un elemento nel data base.

   (**setf** (get $a\ R$) c) pone $R(a, c)$ nel data base, al fondo della lista.

   (**remprop** $a\ R$) toglie ogni $R(a, c)$ dal data base, ed ha lo stesso effetto di (setf (get $a\ R$) NIL).

   (**get** $a\ R$) restituisce il primo c tale che $R(a, c)$ é nel data base attualmente.

   In tutti i casi precedenti, a ed R devono essere atomi (poiché funzionano da nomi), mentre i valori c possono essere liste qualunque.

5 Procedure numeriche

Vengono trattate nel modo solito, usando le funzioni numeriche built-in. Ci sono alcune convenzioni da ricordare:

1. se un argomento di + o − manca, lo si considera 0
2. se un argomento di * o / manca, lo si considera 1
3. se appaiono piú argomenti del necessario, si effettua una iterazione. Ad esempio, (+ 5 10 6) restituisce 21.

6 Logica

1. I **valori di veritá** sono T per true e NIL per false. In generale peró, un qualunque oggetto che non sia valutato a NIL equivale a true.

2. La **negazione** é NOT.

3. La **congiunzione** é AND, che valuta i suoi argomenti da sinistra a destra, e restituisce NIL se uno di essi é NIL, e l'ultimo valore altrimenti. A causa di questo suo comportamento, AND puó essere usato come un condizionale multiplo, ed é utile quando si debbano usare condizionali annidati.

4. La **disgiunzione** é OR, che restituisce il primo argomento di valore non NIL se esso esiste, e NIL altrimenti.

7 Computazioni

Ci sono vari tipi di condizionali. Tutti valutano i loro argomenti soltanto quando devono essere restituiti come valori. Gli argomenti corrispondenti a predicati che falliscono non sono valutati.

1. **if then else**
   Noi sappiamo che if then else é sufficiente, insieme alle funzioni iniziali, successore, predecessore e punti fissi, a generare tutte le funzioni ricorsive (su liste). Esso ha la forma (if primo secondo terzo), e se il primo vale (cioé é valutato non NIL) restituisce il valore del secondo, altrimenti restituisce il valore del terzo.

2. **when**
   (when primo secondo) restituisce il secondo se il primo vale (cioé é valutato non NIL).

3. **unless**
   (unless primo secondo) restituisce il secondo se il primo non vale (cioé é valutato a NIL).

4. **cond**
   Esiste una forma generale di condizionale, che corrisponde ad una definizione per casi multipla. Essa ritorna il valore dell'(ultima) espressione corrispondente al primo predicato che vale (cioé é valutato non NIL), e NIL altrimenti.

Nota che nella definizione si possono avere piú espressioni corrispondenti a ciascun predicato. Solo il valore dell'ultima é restituito, e le altre servono soltanto per avere effetti collaterali (come modificare il valore di certe variabili).

Un altro aspetto dei calcoli é la possibilitá di effettuare programmi strutturati e non, nel modo usuale. A causa della presenza del punto fisso, tale possibilitá é superflua, ed é introdotta soltanto per comoditá.

1. **Programmi non strutturati** (go to)
   La istruzione **prog** dichiara variabili locali, assegna NIL ad esse all'inizio, e reintegra i loro valori correnti alla fine (come per le funzioni).

   L'effetto di prog é quello di saltare atomi che incontra, restituire il valore di una espressione che sia preceduta da **return** (e in questo caso uscire), e ritornare ad un atomo preceduto da un **go**.

   Prog usato con condizionali produce ovviamente flowcharts non strutturati.

2. **Programmi strutturati** (while)
   L'istruzione **do** ha come inputs essenziali due liste.

   La prima lista é costituita di terne, corrispondenti a parametri, valori iniziali di essi, e valori di aggiornamento (da usare in passaggi successivi).

   La seconda lista corrisponde al comportamento da effettuare. Il primo termine é il test per la terminazione, l'ultimo é il risultato da restituire. I termini intermedi servono, al solito, per effetti collaterali, e sono valutati solo se il test ha successo.

   Ci possono poi anche essere espressioni finali, anche queste usate solo per effetti collaterali.

   Il **return** puó essere usato all'interno di un do.

   L'effetto del do é quello di un while, e produce programmi strutturati.

   L'assegnazione dei parametri ad un do é parallela. Per averla seriale, si usa **do***.

8 Predicati built-in

Ce ne sono molti. I piú importanti sono i seguenti.

1. **uguaglianze**
   EQUAL é usato per vedere se due espressioni hanno la stesso valore (cioé puntano allo stesso oggetto), EQ per vedere se sono lo stesso oggetto dal punto di vista della rappresentazione all'interno della macchina. Esse sono ripettivamente l'uguaglianza estensionale (denotare lo stesso oggetto) e intensionale (essere lo stesso oggetto).

2. **dati**
   ATOM, SYMBOLP, NUMBERP, LISTP verificano se un oggetto é del tipo corrispondente.

3. **liste**
   LISTP (essere una lista), ENDP o NULL (essere la lista vuota), MEMBER (essere membro di una lista). Il risultato dell'ultimo predicato é la lista troncata al primo argomento del tipo richiesto che si incontra (questo é un modo particolare di dire true, ma con informazioni aggiuntive).

4. **numeri**
   NUMBERP, ZEROP, PLUSP (essere un numero positivo), MINUSP, $=$, $>$ (essere in ordine decrescente, anche per piú argomenti), $<$.

9 Teoria

Ci sono tre possibili aspetti del Lisp.

1. **computabilitá**
   Le funzioni ricorsive sono calcolabili, quando lo siano le identitá, il successore, il predecessore, la composizione, la definizione per casi e il punto fisso.

   Nel Lisp il punto fisso é implementato automaticamente, per la possibilitá di poter definire funzioni, e di usare il loro nome nella definizione stessa.

La definizione per casi é fornita dalla istruzione if.

Le rimanenti operazioni primitive sulle liste sono definibili da car, cdr e cons.

2. **λ-calcolo**
   Poiché i termini del λ-calcolo si ottengono tutti dalle variabili, per applicazione e λ-astrazione, essi sono traducibili nel Lisp, mediante funcall (o apply) e lambda (le variabili sono gli atomi non numerici).

3. **logica combinatoria**
   I **termini** del Lisp sono gli atomi e le liste. Essi forniscono un modello della logica combinatoria (un'algebra combinatoria totale), interpretando l'applicazione di due termini M e N come la lista $(M\ N)$, e l'uguaglianza di M ed N come vera se (EQUAL M N) restituisce vero (cioé non NIL), dove EQUAL é l'uguaglianza estensionale (M ed N hanno lo stesso valore).

10 Riferimenti

1. Allen, *Anatomy of LISP*, McGraw Hill, 1978.

2. Wilensky, *LISPcraft*, Norton, 1984.

3. Winston e Horn, *Lisp*, Addison Wesley, 1981.