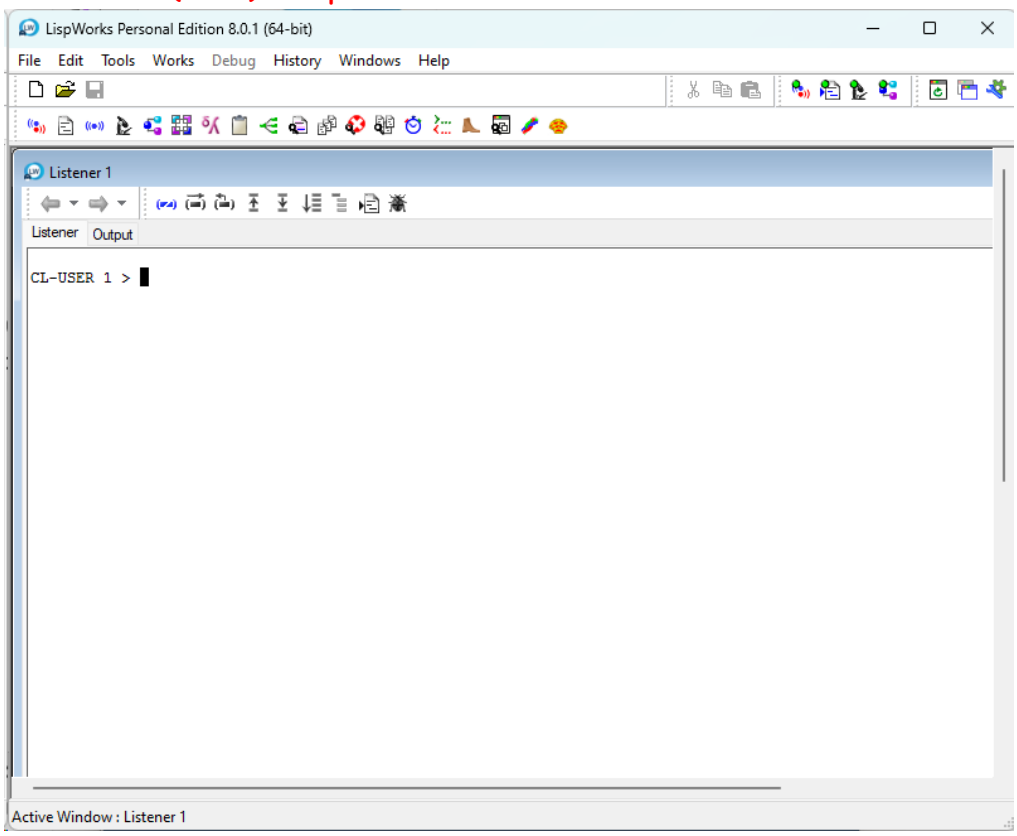
Appunti di Linguaggi di Programmazione

Linguaggio con Paradigma Funzionale – LISP

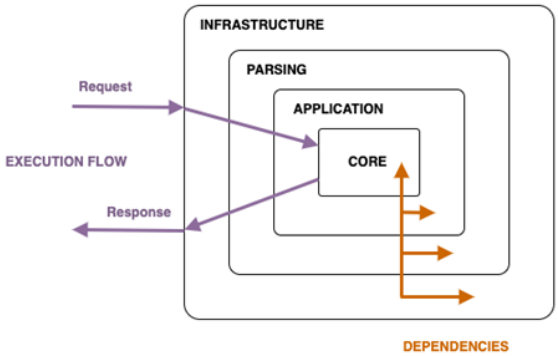
Ambiente (IDE): Lispworks 8.0



Domande e risposte

1) Cosa si intende per programmazione funzionale?

La programmazione funzionale è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche.



2) Definizione di trasparenza referenziale

Si dice trasparenza referenziale la proprietà di una funzione che non ha effetti collaterali e che quando riceve lo stesso parametro in input, restituisce sempre lo stesso valore. Gli effetti collaterali sono una delle caratteristiche negative dei linguaggi imperativi ovvero comportano modifiche dello stato di memoria di un programma.

3) Caratteristiche della programmazione funzionale

Nel paradigma funzionale gli oggetti di vario tipo e strutture di controllo vengono raggruppati logicamente in modo diverso da come invece accade nel paradigma imperativo. In particolare è utile pensare in termini di:

- espressioni (funzioni primitive e non);
- modi di combinare tali espressioni per ottenerne di più complesse (composizione);
- modi e metodi di costruzione di "astrazioni" per poter far riferimento a gruppi di espressioni per "nome" e per trattarle come unità separate;
- operatori speciali (condizionali).

4) Definizione di funzione

Si dice funzione, una regola per associare gli elementi di un insieme (dominio, domain) a quelli di un altro insieme (codominio, range). Essa può essere applicata a un elemento del dominio e restituisce un elemento del codominio detto valore.

Esempio:

```
quadrato(x) ≡ x * x
```

x è l'argomento della funzione, mentre x * x è il valore della funzione.

5) Composizione di funzioni

Nei linguaggi funzionali espressioni più complesse vengono costruite mediante composizione. Se F è definita come composizione di G e H:

$$F \equiv G \circ H$$

applicare F equivale ad applicare G al risultato dell'applicazione di H.

$$alla\_quarta \equiv quadrato \circ quadrato$$

6) Linguaggio LISP: caratteristiche

LISP (LIST Processing) è famiglia di linguaggi funzionali con costrutti imperativi di "convenienza". Le caratteristiche di LISP sono:

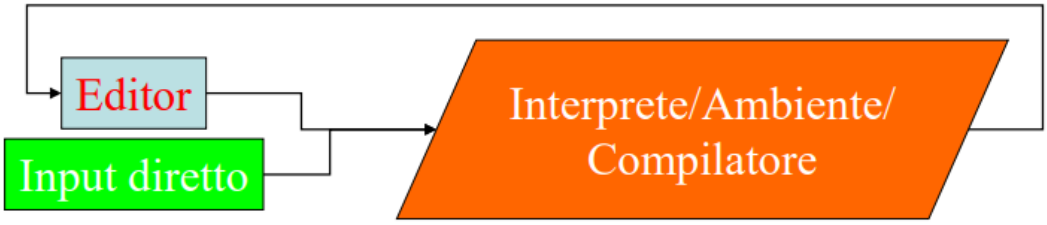
→ è un linguaggio interpretato, come Javascript;

→ le espressioni più semplici sono numeri e stringhe;

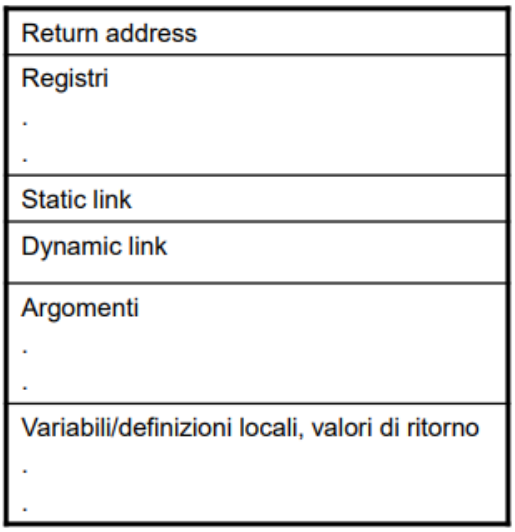
→ le operazioni elementari possono essere scritte come +(x, y), come ad esempio +(1, 5), -(44, 2). In LISP, la notazione diventa (f x_1 x_2 … x_n) con parentesi e spazi tra gli argomenti;
→ l'ordine di valutazione consiste nel partire da sinistra verso destra, applicando f con i vari argomenti;
→ non esistono costrutti come while e for ed è un linguaggio basato totalmente sulla ricorsione combinata con gli operatori speciali.

7) Interpreti, ambienti e compilatori e activation frame

In LISP e vari altri linguaggi di programmazione spesso si interagisce invece con un "ambiente" (spesso contenente il compilatore) la cui interfaccia principale è un interprete.



L'activation frame è una struttura dati che viene costruita sullo stack di sistema per valutare delle funzioni e i parametri formali di una funzione vengono associati ai valori (non esistono effetti collaterali e tutto viene passato per valore). Ad ogni sotto-espressione del corpo si sostituisce il valore che essa denota (computa) e il valore (valori) restituito dalla funzione è il valore del corpo della funzione (che non è altro che una sotto-espressione).
Una activation frame è costituita da:
→ valore di ritorno (return address);
→ registri;
→ static link (riferimento statico);
→ dynamic link (riferimento dinamico);
→ argomenti;
→ valori di ritorno;
→ variabili e definizioni locali.
Attenzione: static link e dynamic link sono molto importanti perchè servono a mantenere informazioni circa il dove una funzione è definita e quando una funzione viene chiamata.



8) Espressioni in LISP

In LISP ogni "espressione" denota un "valore" e le espressioni più semplici sono numeri e stringhe. Infatti se nell'interprete vengono inseriti numeri e stringhe si ha il seguente risultato:

```
CL-USER 8 : 1 > "ciao"
"ciao"

CL-USER 9 : 1 > (+ 32 10)
42

CL-USER 10 : 1 > (* 24 24)
576
```

9) Programmazione funzionale in LISP

Le operazioni aritmetiche elementari +, -, * e / non sono altro che funzioni. Questo tipo di notazione per le operazioni aritmetiche elementari si dice notazione prefissa.

```
CL-USER 9 : 1 > (+ 32 10)
42

CL-USER 10 : 1 > (* 24 24)
576

CL-USER 11 : 1 > (/ 23 2)
23/2

CL-USER 13 : 2 > (/ 24 2)
12
```

Le funzioni aritmetiche in LISP accettano un numero variabile di argomenti.

```
CL-USER 14 : 2 > (/ 24 2 3 4)
1
```

Le espressioni possono essere anche più complicate.

```
CL-USER 1 > (+ (* 3 (+ (* 2 4) (+ 3 2))) (+ (- 10 8) 1))
42
```

Si nota anche che le funzioni aritmetiche elementari + e * in (Common) LISP rispettano i vincoli di "campo" algebrico.

```
CL-USER 13 : 7 > (+)
0

CL-USER 14 : 7 > (*)
1
```

10) Funzioni e "costanti" in LISP

In LISP si hanno :

→ numeri interi: 42, -3;

→ numeri virgola mobile: 0.5, 3.14, 6.02E+21;

→ numeri razionali: 3/2, -3/42;

→ numeri complessi: #C(0 1);

→ booleani (*): T NIL;

→ stringhe: "stringa";

→ booleani: null, and, or e not;

→ funzioni su numeri: +, -, /, *, mod, sin, cos, sqrt ed ecc.

11) Ordine di valutazione

Data un'espressione LISP

$(f\ x_1\ x_2\ \ldots\ x_N)$

la valutazione procede da sinistra verso destra a partire da x_1 fino a x_N producendo l valori v_1, …, v_N e questa regola è inerentemente ricorsiva. Alcuni operatori speciali valutano gli argomenti in modo diverso e questi sono ad esempio if, cond, defun, defparameter, quote, etc.

12) Definizione di variabili e di funzioni

In Common LISP, le variabili vengono definite utilizzando l'operatore speciale **defparameter**.

```
CL-USER 1 > (defparameter quarantadue 42)
QUARANTADUE
```

Si ha che quarantadue ha ora associato il valore 42.

Le funzioni si possono definire usando l'operatore speciale **defun**.

Si ha quindi:

```
CL-USER 2 > (defun quadrato (x) (* x x))
QUADRATO
```

dove:

defun → keyword;

quadrato → nome funzione;

(x) → lista dei parateri formali;

(* x x) → corpo della funzione.

L'operatore **defun** associa il corpo della funzione al nome nell'ambiente globale

del sistema Common LISP e restituisce come valore il nome della funzione.

Esempi:

```
CL-USER 1 > (defparameter quarantadue 42)
QUARANTADUE

CL-USER 2 > (defun quadrato (x) (* x x))
QUADRATO

CL-USER 3 > (quadrato quarantadue)
1764

CL-USER 4 > (- (quadrato (+ 20 22)) 10)
1754

CL-USER 5 > (defun somma_quadrati (x y) (+ (quadrato x) (quadrato y)))
SOMMA_QUADRATI

CL-USER 6 > (- (somma_quadrati 6 3) 3)
42
```

13) Nomi ed identificatori in (Common) LISP

In (Common) LISP i nomi possono contenere il carattere '-' e ciò è possibile perché non c'è ambiguità con il segno '-' ovvero con la funzione '-'.

Esempi:

```
a b c quarantadue franklin-delano-roosevelt
barak-hussein-obama lizzie2020 unico2018 x $32
%42 ford_prefect vogonian@poetry.com
```

14) Valutazione di funzioni

La valutazione di funzioni avviene mediante la costruzione di activation frames.

Le caratteristiche della modalità di valutazione di funzioni sono:

→ i parametri formali vengono associati ai valori (tutto si passa per valore);

→ ad ogni sotto espressione del corpo si sostituisce il valore che essa denota;

→ il valore/i restituito dal corpo della funzione è il valore del corpo della funzione e quando il valore finale viene ritornato, l'activation frame viene rimosso.

15) Funzioni anonime (operatore lambda)

Il linguaggio LISP ammette la creazione a runtime di funzioni, grazie alle strutture closures. Si dice operatore lambda, l'operatore che permette la costruzione di funzioni anonime e mediante ciò è possibile creare delle funzioni senza assegnare loro un nome.

```
CL-USER 9 > (lambda (x1 x2 xN) <e>)
```

(x_1 x_2 … x_n) → parametri

‹e› → espressione

Esempio:

```
CL-USER 1 > (lambda (x) (+ x 42))
#<anonymous interpreted function 40300128DC>

CL-USER 2 > ((lambda (x) (+ x 42)) 42)
84
```

16) Operatori condizionali e booleani

L'operatore condizionale in LISP (cond) permette di assegnare un valore finale alla funzione a seconda della verità o meno della condizione preliminare.
L'operatore condizionale ha la seguente sintassi: `(cond (c1 e1) (c2 e2) (cm em))`
Le funzioni sono dei "predicati" che ritornano i valori "vero" o "falso"; la costante T rappresenta il valore di verità "vero"; la costante NIL rappresenta il valore di verità "falso".

Esempio:

```
CL-USER 4 : 1 > (= 42 0)
NIL

CL-USER 5 : 1 > (> 42 0)
T
```

Come in ogni linguaggio, anche il (Common) LISP ha a disposizione i soliti operatori booleani, che appaiono a tutti gli effetti come delle funzioni

(and c_1 c_2 … c_k)
(or d_1 d_2 … d_k)
(not e)

Esempio

```
CL-USER 6 : 1 > (and (> 42 0) (< -42 0))
T

CL-USER 7 : 1 > (not (> 42 0))
NIL

CL-USER 8 : 1 > (and)
T

CL-USER 9 : 1 > (or)
NIL
```

17) Funzioni ricorsive e definizione di tail recursion

Un esempio tipico di funzione ricorsiva è il fattoriale:

```
CL-USER 10 : 1 > (defun fattoriale (n)
                   (if (= n 0) 1
                        (* n (fattoriale (- n 1)))))
FATTORIALE
```

Le funzioni ricorsive possono essere distinte in due categorie: funzioni ricorsive in testa e funzioni ricorsive in coda.

Nella ricorsione in testa la chiamata ricorsiva rappresenta la prima operazione della procedura, mentre nella ricorsione in coda la chiamata ricorsiva rappresenta l'ultima operazione eseguita dalla funzione chiamante. Questa differenza è fondamentale riguardo l'ottimizzazione del codice da parte del compilatore. Una funzione ricorsiva in coda può essere trasformata in un semplice loop, eliminando completamente la necessità di costruire activation frame, effettuare operazioni di push nello stack, passare il controllo alla funzione chiamata, riottenere il controllo, effettuare l'operazione di pop dell'activation frame. Questa tipologia di ricorsione risulta quindi più performante.

Ricorsione in testa

```
(defun fattoriale(n)
    (if (= n 0)
     1
     (* n (fattoriale (- n 1)))))
```

Ricorsione in coda

```
(defun fattoriale(n acc)
    (if (= n 0)
     acc
     (fattoriale (- n 1) (* n acc))))
```

18) Strutture dati e CONS – CELLS

Si dice cons – cells, una struttura dati di LISP costituita da una coppia di puntatori a due elementi. Le cons – cells sono create dalla funzione cons, che alloca la memoria necessaria al mantenimento della struttura: vengono generati in memoria dei grafi di puntatori.

La funzione cons è così definita:

cons : *<oggetto Lisp>* × *<oggetto Lisp>* → *<cons-cell>*

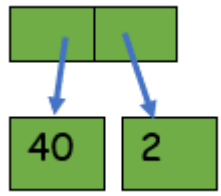
I due puntatori di una cons – cell sono chiamati - per ragioni storiche - car e cdr, a cui corrispondono due funzioni.

Esempio:

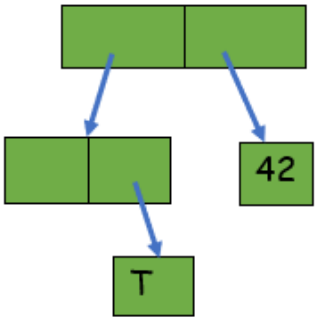
```
CL-USER 1 > (defparameter c (cons 40 2))
C

CL-USER 2 > (car c)
40

CL-USER 3 > (cdr c)
2
```

Rappresentazione grafica

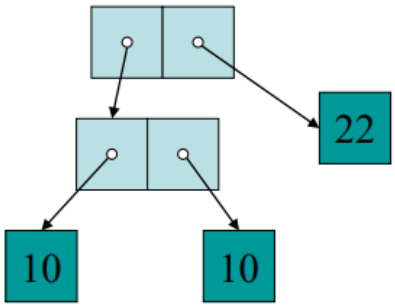
1 - `(cons 40 2)`



2 - `(cons (cons NIL T) 42)`



3 - `(cons (cons 10 10) 22)`



Le risposte dell'interprete LISP sono:

```
CL-USER 6 > (cons (cons 10 10) 22)
((10 . 10) . 22)

CL-USER 7 > (cons NIL T)
(NIL . T)

CL-USER 8 > (cons 4 2)
(4 . 2)

CL-USER 9 > (cons "foo" 42)
("foo" . 42)
```

19) Liste e funzioni su liste

La funzione cons può quindi essere usata per rappresentare sequenze (liste) di oggetti.

Esempio:

Sia L, la lista così definita

```
CL-USER 10 > (defparameter L (cons 1 (cons 2 (cons 3 (cons 4 NIL)))))
L
```

l'istruzione genera in memoria una sequenza di cons – cells tale che

```
CL-USER 11 > (car (cdr (cdr L)))
3
```

Si ha la possibilità di costruire la lista nel seguente modo alternativo:

```
CL-USER 1 > (defparameter L (list 1 2 3 4))
L
```

La funzione list accetta un numero variabile di argomenti. Si avrebbe quindi:

```
CL-USER 2 > (list -1 0 1 2 3)
(-1 0 1 2 3)
```

Non devono essere confuse dunque le due espressioni:

→ `(list 1 2 3)`: indica un'espressione a tutti gli effetti;

→ `(1 2 3)`: rappresenta tipograficamente una struttura dati in memoria.

Risulta utile anche notare

```
CL-USER 5 : 1 > (list)
NIL
```

I possibili algoritmi sulle liste sono

1 – estrazione n – esimo elemento da una lista

```
(defparameter l (list 1 2 3 4 5 6 7))

(defun extract_n_el_list (l n)
  (if (<= n 0)
      (car l)
      (extract_n_el_list (cdr l) (- n 1))))
```

Output

```
CL-USER 6 : 1 > (extract_n_el_list l 4)
5

CL-USER 7 : 1 > (extract_n_el_list l 0)
1

CL-USER 8 : 1 > (extract_n_el_list l 1)
2

CL-USER 9 : 1 > (extract_n_el_list l 8)
NIL
```

2 – lunghezza lista

```
(defparameter l (list 1 2 3 4 5 6 7 8 9 10))

(defun lunghezza_lista (l)
  (if (not (car l))
      0
      (+ 1 (lunghezza_lista (cdr l)))))
```

Output

```
CL-USER 10 : 1 > (lunghezza_lista l)
10
```

3 – unione due liste

```
(defun append_even_odd_list (l1 l2)
  (if (null l1)
      odd
      (cons (car l1) (append even odd list (cdr l1) l2))))
```

20) Operatore QUOTE: definizione

Si definisce quote (quote <e>), l'operatore che non permette la valutazione di una determinata espressione definita in <e>.

Esempio:

```
CL-USER 21 : 5 > (quote (+ 1 2))
(+ 1 2)
```

21) Espressioni autovalutanti

Le espressioni autovalutanti sono particolari espressioni il cui valore è semplicemente quello espresso dalla propria rappresentazione. Ciò implica che stringhe, numeri, booleans (T e NIL) e lambda expression sono tutte espressioni autovalutanti. L'interprete LISP non deve quindi chiamare alcuna funzione o attuare alcuna trasformazione per comprenderne il valore associato.

22) Tipi di dati in (Common) LISP

In LISP si ha una ripartizione degli oggetti principali in due categorie: atomi e cons – cells. Gli atomi sono simboli e numeri (anche stringhe) mentre le cons – cells non sono atomi. Esiste un predicato che permette di stabilire se l'argomento è un predicato o meno: atom.

```
CL-USER 24 : 6 > (atom "ero")
T

CL-USER 25 : 6 > (atom 'ero)
T

CL-USER 26 : 6 > (atom 1)
T

CL-USER 27 : 6 > (atom (cons 42 42))
NIL
```

In LISP si hanno quindi:

→ numeri;

→ simboli;

→ stringhe;

→ le liste (cons – cells).

Le cons – cells più numeri, simboli e stringhe costituiscono quelle che si chiamano Symbolic Expressions (dette anche sexp's).

23) Valutazioni di funzioni ricorsive

Dato che programmi e sexp's in LISP sono equivalenti è possibile valutare una funzione mediante la funzione eval.
Data una sexp:
→ se è un atomo (non lista), allora se questa è una stringa o numero restituisce tale valore, altrimenti se è un simbolo estrae il suo valore dall'ambiente corrente e lo restituisce e in caso contrario segnala errore;
→ se è una cons – cell $(O\ A_1\ A_2\ \dots\ A_n)$, allora si procede nel seguente modo:
a) se O è un operatore speciale, allora lo si valuta come operatore speciale;
b) se O è un simbolo che denota una funzione nell'ambiente corrente, allora questa funzione viene applicata (mediante apply) alla lista (VA1, VA2, …, VAn) che raccoglie i valori delle valutazioni delle espressioni (A1, A2, …, An);
c) se O è una Lambda Expression, allora questa viene applicata alla lista che raccoglie le valutazioni delle espressioni (A1, A2, …, An);
d) altrimenti viene segnalato errore.

24) Uguaglianza

Esistono varie espressioni di uguaglianza. Quelle più note sono:
→ eql: controlla l'uguaglianza di simboli e numeri interi;
→ equal: come eql, ma è in grado di controllare se le due liste sono uguali.

25) Funzioni di ordine superiore

Vi sono diverse funzioni o primitive di ordine superiore:
→ mapcar: applica la funzione f a tutti gli elementi della lista L e ritorna una lista dei valori;
→ let: ci permette di introdurre nuovi nomi (variabili) locali da poter riutilizzare all'interno di una procedura;
→ compose: corrisponde alla composizione di funzioni ritornando una nuova funzione f(g(x));
→ filter: rimuove tutti gli elementi della lista che non soddisfano tale predicato;
→ fold (reduce): applica una funzione ad un elemento di una lista ed al risultato (ricorsivo) dell'applicazione di reduce al resto della lista.

26) Parametri opzionali, lunghezza variabile ed a chiave

In (Common) LISP certi simboli hanno un'interpretazione particolare. Ad esempio, il carattere ":" rappresenta una keyword e hanno sé stessi come valore.

```
cl-prompt> :foo
:foo

cl-prompt> :forty-two
:forty-two
```

È possibile anteporre un indicatore di parametro all'argomento, durante la definizione di funzioni, che ne modifica il carattere:
- &rest rende il parametro variabile
- &optional rende il parametro opzionale
- &key rende il parametro a chiave.

Una lista di argomenti variabili è chiamata lambda lists e si definisce con l'indicatore &rest.
I parametri opzionali, a chiave e variabili vanno sempre dichiarati dopo quelli obbligatori.

27) Streams I/O e File I/O LISP

La gestione I/O in LISP viene effettuata principalmente con due primitive: read e print. Le caratteristiche di tale gestione sono:
→ read: legge in intero oggetto LISP;
→ print: stampa rispettando la sintassi;
→ format: formatta l'output;
→ il carattere "t" è lo standard output, uno dei 3 stream standard (con input error);
→ la scrittura e la lettura avviene su uno stream;
→ with – open – file: associa uno stream ad un file.

28) Caratteristiche di REPL

Le tre fasi che permettono di applicare il paradigma funzionale sono READ, EVAL, PRINT. In particolare, l'ambiente di programmazione viene chiamato REPL, perché è un ciclo (LOOP) che ad ogni interazione accetta input da parte dell'utente (READ), li valuta (EVAL) e restituisce il risultato all'utente (PRINT). Di questo tipo è il sistema LISP.

29) Funzioni Apply ed Eval

Si definisce funzione apply, la funzione tale che apply: funzione list → exp, ovvero prende un designatore di funzione e restituisce un valore.
Si definisce funzione eval, la funzione tale che eval: sexp env → sexp.
Un ambiente e una sequenza di frame, e un frame è una lista di coppie prefissa dal simbolo frame.

30) Interprete meta – circolare

Si dice interprete meta – circolare, l'interprete LISP, diviso in:

→ ENV: operazioni riguardo alla manipolazione di frames e environments;

→ IMC: interprete "vero e proprio";

→ REPL: riguardo alle tre fasi che permettono di applicare il paradigma funzionale (READ – EVAL – PRINT LOOP).

31) Espressione Lambda: definizione

Espressione anonima, creata a runtime, grazie alle closures, che permette di creare diverse funzioni senza dover assegnare loro un nome.

32) Definizione di chiusura

La chiusura è una struttura dati generata da una lambda e che contiene il corpo della lambda, la lista dei parametri formali e l'ambiente in cui è stata costruita.