Linguaggi di Programmazione 2022-2023
Lisp e programmazione funzionale II

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

Indice

- Nomi ed identificatori in (Common) Lisp
- Valutazione di funzioni
- Funzioni anonime
- Operatori condizionali e booleani
- Funzioni ricorsive
- Strutture dati e `CONS`
- Liste e funzioni su liste

Preliminare: nomi in (Common) Lisp

- Come avete potuto notare, in (Common) Lisp i nomi possono contenere il carattere ‘–’
- Questo perchè non c’è ambiguità con il segno ‘–’ ovvero con la funzione ‘–’
- In generale i **simboli** (nomi) LISP sono sequenze alfanumeriche aumentate con qualunque carattere esclusi **(**, **)**, **#**, **`**, **’**, **”**, **:** e la virgola
- Esempi

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

Valutazione di funzioni: introduzione

- La valutazione di funzioni avviene mediante la costruzione (sullo stack di sistema) di activation frames
- I parametri formali di una funzione vengono associati ai valori (nb: si passa tutto per **valore**; ribadiamo che *non esistono effetti collaterali*)
- Il corpo della funzione viene valutato (ricorsivamente) tenendo conto di questi legami in maniera statica
  - Ovvero bisogna tener presente cosa accade con variabili che risultano "libere" in una sotto-espressione
- Ad ogni sotto-espressione del corpo si sostituisce il valore che essa denota (computa)
- Il valore (valori) restituito dalla funzione è il valore del corpo della funzione (che non è altro che una sotto-espressione)
  - Quando il valore finale viene ritornato i legami temporanei ai parametri formali spariscono (lo stack di sistema subisce una "pop" l'activation frame viene rimosso)

Valutazione di funzioni: introduzione

- Esempio

```
prompt> (defun doppio (n) (* 2 n))
doppio
```

  - estende l'ambiente globale con il legame tra **`doppio`** e la sua definizione

```
prompt> (doppio 3)
6
```

  - estende l'ambiente globale con quello locale che contiene i legami tra parametri formali e valori dei parametri attuali
    - un activation frame viene inserito in cima allo stack di valutazione
  - valuta il corpo della funzione
  - ripristina l'ambiente di partenza
    - l'activation frame viene rimosso dalla cima dello stack di valutazione

Chiamate a funzioni: richiamo

- Per la valutazione di una funzione **F**, l'ambiente deve eseguire i seguenti sei passi:
  - mettere i parametri in un posto dove la procedura possa recuperarli
  - trasferire il controllo alla procedura
  - allocare le risorse (di memorizzazione dei dati) necessarie alla procedura
  - effettuare la computazione della procedura
  - mettere i risultati in un posto accessibile al chiamante
  - restituire il controllo al chiamante

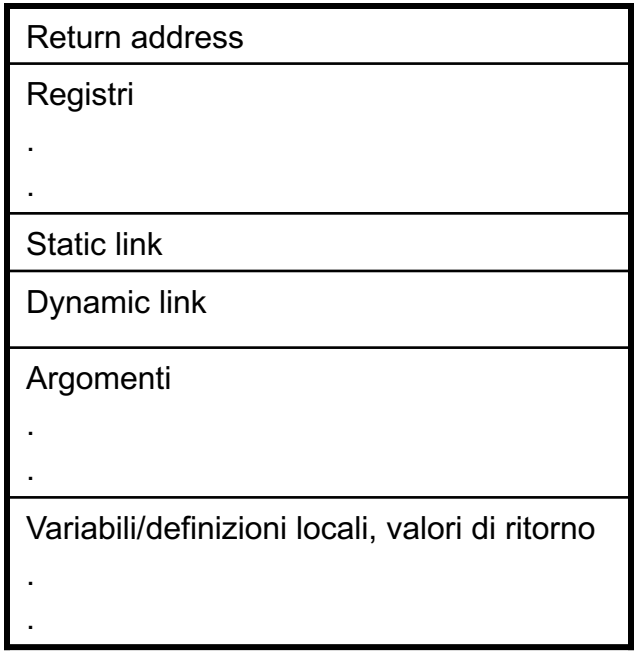
- Queste operazioni agiscono sui registri a disposizione e sullo "stack" utilizzato dal runtime (esecutore) del linguaggio

- Lo spazio richiesto per salvare (sullo stack) tutte le informazioni necessarie all'esecuzione di una funzione **F** ed al ripristino dello stato precedente alla chiamata è quindi costituito da
  - Spazio per i registri da salvare prima della chiamata di una sotto funzione
  - Spazio per l'indirizzo di ritorno (nel codice del corpo della funzione)
  - Spazio per le variabili, definizioni locali, e valori di ritorno
  - Spazio per i valori degli argomenti
  - Spazio per il riferimento statico (*static link*)
  - Spazio per il riferimento dinamico (*dynamic link*)
  - Altro spazio dipendente dal particolare linguaggio e/o politiche di allocazione del compilatore

Chiamate di funzioni: riassunto

Activation Frame di una funzione

- Spazio per i registri da salvare prima della chiamata di una sotto funzione
- Spazio per l'indirizzo di ritorno (*return address* nel codice del corpo della funzione)
- Spazio per le variabili, definizioni locali e spazio per valori di ritorno
- Spazio per i valori degli argomenti
- Spazio per il riferimento statico (*static link*)
- Spazio per il riferimento dinamico (*dynamic link*)
- Altro spazio dipendente dal particolare linguaggio e/o politiche di allocazione del compilatore



Chiamate di funzioni: riassunto

- **Esempio**

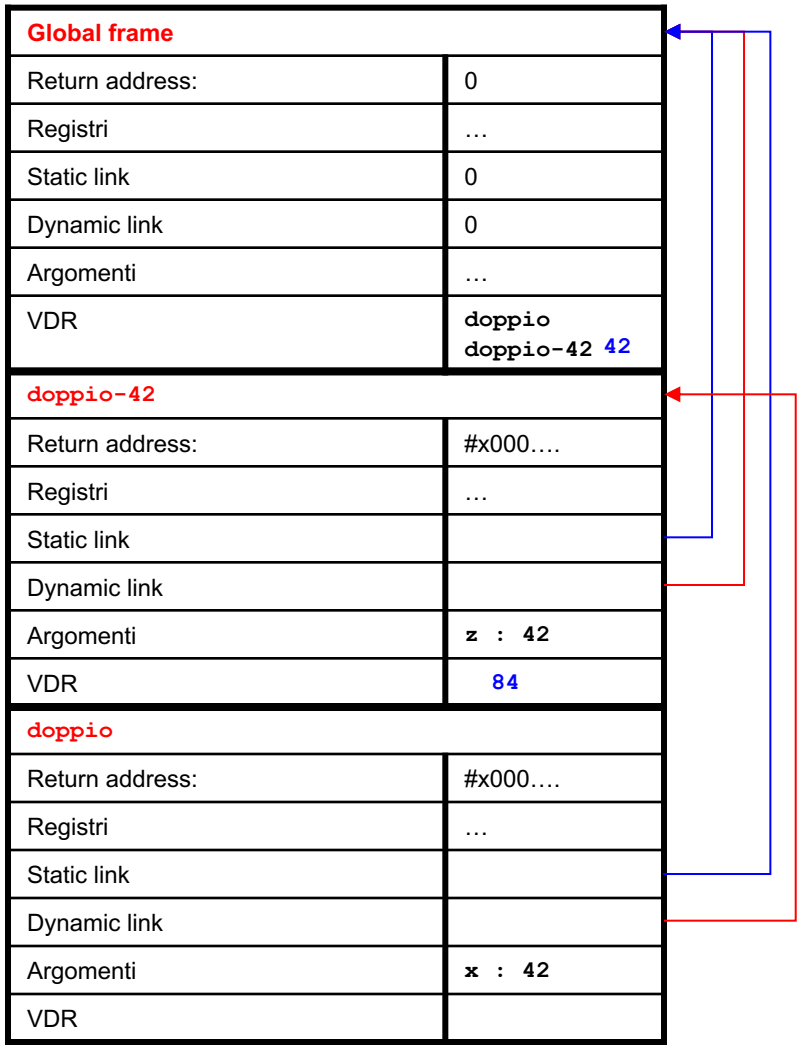
```
(defun doppio (x) (* 2 x))

(defun doppio-42 (z) (- (doppio z) 42))
```

La chiamata `(doppio-42 42)` ha il seguente effetto:

- **Static link** e **dynamic link** sono molto importanti perchè servono a mantenere informazioni circa il
  - Dove una funzione è definita
  - Quando una funzione è chiamata

- Il contenuto effettivo di un activation frame dipende da diverse scelte implementative

- L'esempio riportato non è necessariamente completo



Creazione di funzioni in Lisp

- La gestione dello static link in Common Lisp è in realtà più complicato, dato che il linguaggio ammette *la creazione a runtime* di funzioni che si devono ricordare il valore delle loro variabili libere al momento della loro creazione

- Queste funzioni sono implementate con particolari strutture dati chiamate **chiusure** (***closures***)
  - "*chiudono*" i valori delle loro variabili libere

- Questa caratteristica del Common Lisp (e dei linguaggi funzionali in genere) ci permette di creare delle funzioni **anonime**

  ⇒ operatore **`LAMBDA`**

Funzioni anonime: espressioni LAMBDA

- Una delle caratteristiche dei linguaggi funzionali (e del Lisp in particolare) è la capacità di costruire funzioni anonime; l'operatore che ci permette di costruire funzioni anonime si chiama **LAMBDA**

- L'operatore **lambda** (il cui nome deriva dal λ-calcolo) denota una funzione anonima la cui sintassi è la seguente

  (**lambda** (x_1 x_2 … x_N) <e>)

  dove x_1 x_2 … x_N sono dei simboli che rappresentano i parametri formali ed <e> è un'espressione

- Queste espressioni sono chiamate **lambda**-espressioni (in Inglese **lambda**-*expressions*)

Funzioni anonime: espressioni LAMBDA

- Esempio

```
(lambda (x) (+ 2 x))
```

è la funzione che aggiunge **2** ad **x**

```
((lambda (x) (+ 2 x)) 40)
```

è l'**applicazione** della funzione anonima al numero **40**; il risultato è, ovviamente, **42**!

- Una lambda-expression può essere usata ovunque possiamo usare un nome di una funzione

Funzioni anonime ed operatore `lambda`

- Con l'operatore lambda possiamo creare tutte le funzioni che vogliamo senza assegnare loro un nome

```
prompt> (lambda (x) (+ x 42))
#<funzione>

prompt> ((lambda (x) (+ x 42)) 42)
84
```

Operatori speciali: condizionali

- Supponiamo di voler definire una funzione chiamata

  `valore_assoluto`

  che trasformi un numero nel suo valore assoluto; la definizione matematica è la seguente

$$\texttt{valore\_assoluto}(x) = \begin{cases} x & \text{se} \quad x > 0 \\ 0 & \text{se} \quad x = 0 \\ -x & \text{se} \quad x < 0 \end{cases}$$

Operatori speciali: condizionali

- Questo tipo di definizione matematica viene detta “per casi”, ovvero, si danno una serie di casi dove il valore finale della funzione dipende dalla verità o meno della condizione preliminare
- In LISP questo tipo di costrutto è disponibile tramite l’operatore speciale `cond`
- **`cond`** ha la seguente sintassi

  (**cond** (c_1 e_1) (c_2 e_2) … (c_M e_M))

  dove c_1, …, c_M ed e_1, …, e_M sono normali espressioni; le parentesi sono obbligatorie dato che delimitano coppie di espressioni

Operatori speciali: condizionali

- Date le funzioni **<**, **=**, e **>**, la funzione **`valore-assoluto`** può quindi essere definita nel modo seguente

```
(defun valore-assoluto (x)
   (cond ((> x 0) x)
         ((= x 0) 0)
         ((< x 0) (- x))))
```

- Le funzioni **<**, **=**, e **>** 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"

```
prompt> (= quarantadue 0)
NIL

prompt> (> quarantadue 0)
T
```

Operatori speciali: condizionali

- L'operatore **`cond`** viene valutato in maniera speciale
- Ogni coppia $(c_j\ e_j)$ viene considerata in ordine
  - Se c_i ha valore `T` allora il valore ritornato dalla valutazione dell'operatore `cond` è il valore ottenuto dalla valutazione di e_j
  - Altrimenti la valutazione considera la coppia successiva $(c_{j+1}\ e_{j+1})$
  - Se non vi sono più coppie allora **`cond`** produce il valore `NIL`

```
prompt> (valore-assoluto 3)
3

prompt> (valore-assoluto -42)
42
```

Operatori speciali: condizionali

- Naturalmente la funzione **`valore-assoluto`** può essere definita anche come

```
(defun valore-assoluto (x)
    (cond ((> x 0) x)
          (T (- x))))
```

- Questo tipo di definizioni è così comune che il (Common) Lisp offre l'abbreviazione **`if`**, con la sintassi

(**`if`** c e_1 e_2)

- La funzione **`valore-assoluto`** diventa

```
(defun valore-assoluto (x)
   (if (> x 0) x (- x)))
```

Operatori speciali: booleani

- 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 c1 c2 … cK)
(or d1 d2 … dM)
(not e)
```

La loro semantica è la solita: ecco alcuni esempi

```
prompt> (and (> 42 0) (< -42 0))
T

prompt> (not (> 42 0))
NIL
```

Operatori speciali: booleani

- Anche in questo caso vengono rispettate alcune relazioni fondamentali

```
prompt> (and)
T

prompt> (or)
NIL
```

Un esempio: radici quadrate con il metodo di Newton

- Come possiamo costruire una funzione che calcoli la radice quadrata di un numero? Il metodo più immediato è di fare una serie di approssimazioni successive utilizzando il metodo di Newton
- Ad esempio, per calcolare la radice di 2 possiamo fare i seguenti calcoli partendo da un valore congetturato

Congettura	Quoziente	Media
1	(2/1) = 2	((2 + 1)/2) = 1.5
1.5	(2/1.5) = 1.3333	((1.3333 + 1.5)/2) = 1.4167
1.4167	(2/1.4167) = 1.4118	((1.4167 + 1.4118)/2) = 1.4142
1.4142		

Un esempio: radici quadrate con il metodo di Newton

- Ciò ci suggerisce un modo per definire la nostra funzione a partire da un numero congetturato *c*

```
(defun ciclo-radice-quadrata (x c)
    (if (va-bene? x c) c
        …))
```

Ovvero se c va bene come radice quadrata di **x** allora tanto vale ritornarlo come risultato

- Ovviamente abbiamo bisogno di definire la funzione **va-bene?**

```
(defun va-bene? (x c)
    (< (valore-assoluto (- x (quadrato c)))
       0.001))
```

Un esempio: radici quadrate con il metodo di Newton

Cosa si fa se *c* non va bene?

1. Si cerca di *migliorare* la congettura (utilizzando la media, secondo lo schema spiegato precedentemente)

```
(defun media (x y) (/ (+ x y) 2.0))

(defun migliora (c x) (media c (/ x c)))
```

2. … e si ripete la procedura, finchè non si trova un valore *c* che **va-bene?**

```
(defun ciclo-radice-quadrata (x c)
    (if (va-bene? x c)
        c
        (ciclo-radice-quadrata x (migliora c x))))
```

Un esempio: radici quadrate con il metodo di Newton

- Dato che il numero **1** va sempre bene (per radici di numeri maggiori od uguali ad 1) come congettura iniziale possiamo completare la nostra definizione come

```
(defun radice-quadrata (x)
     (ciclo-radice-quadrata x 1.0))
```

- È ovvio che **`ciclo-radice-quadrata`** è una funzione **ricorsiva**

- È anche noto che tutti gli algoritmi che richiedono dei cicli possono essere trasformati in un insieme di funzioni ricorsive

- ***Il contrario non è vero!***

Funzioni ricorsive

- Un altro esempio tipico di funzione ricorsiva è il fattoriale

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

- Questa funzione calcola il fattoriale di un numero utilizzando una sequenza di valori intermedi che devono essere salvati da qualche parte (ovvero sullo “stack” di attivazione di ogni chiamata ricorsiva)

Funzioni ricorsive

- Un altro tipico esempio di funzione ricorsiva è quella che calcola l'ennesimo numero di Fibonacci

```
(defun fib (n)
     (cond ((= n 0) 1)
           ((= n 1) 1)
           (T (+ (fib (- n 2)) (fib (- n 1))))
            ))
```

- Questa funzione ha una struttura di chiamate ricorsive radicalmente diversa da quella della funzione fattoriale

Funzioni ricorsive

- La funzione fattoriale può essere riscritta nel seguente modo

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

(defun fattoriale (n)
     (fatt-ciclo n 1))
```

- Ovvero la funzione fattoriale può essere riscritta in modo tale da riutilizzare uno degli argomenti come un *accumulatore*
- La funzione **`fib`** non può essere direttamente riscritta in tale maniera

Funzioni ricorsive

- La funzione **fatt-ciclo** (e la funzione **ciclo-radice-quadrata**) non sono altro che dei cicli in incognito

- Questo tipo di funzioni ricorsive è particolare poichè un compilatore può ottimizzare la chiamata ricorsiva con un'operazione di JUMP, senza creare un nuovo record di attivazione

- Queste funzioni vengono dette funzioni **tail-ricorsive**

- La funzione **`fib`** non può essere direttamente riscritta in modo tail-ricorsivo poichè richiede la combinazione di **due** chiamate ricorsive

Ricorsione

- Il concetto di ricorsione è fondamentale in informatica e matematica
- Un insieme di funzioni mutualmente ricorsive può rappresentare una macchina di Turing
  - Cfr. Tesi di Church e Kleene
- Quindi, i linguaggi funzionali puri (senza assegnamenti e “salti”) sono Turing-completi
- I LISP originari erano essenzialmente puri, anche se il primo LISP diffuso (LISP 1.5, McCarthy http://www-formal.stanford.edu/jmc/history/lisp/lisp.html) già conteneva elementi imperativi
  - Fu solo con la serie di articoli di Steele (LAMBDA The Ultimate:… 1976 http://lambda-the-ultimate.org/node/4) che buona parte della distanza tra teoria e pratica fu azzerata

Strutture Dati e Funzioni

- Supponiamo di voler costruire una piccola libreria per fare calcoli con numeri razionali (*)
- Innanzitutto assumiamo di aver a disposizione una funzione che costruisce una *rappresentazione* di un numero razionale

  `(crea-razionale n d)` $\Rightarrow$ **<il razionale *n*/*d*>**

  dove *n* è il numeratore e *d* è il denominatore
- Assumiamo anche di avere due funzioni **`numer`** and **`denom`** che estraggono rispettivamente il numeratore *n* ed il denominatore *d* dalla rappresentazione di un numero razionale <razionale `n/d`>
- Costruire la libreria è a questo punto semplicissimo

(*) Il Common Lisp e Scheme possono manipolare direttamente i numeri razionali; la libreria presentata serve solo a scopi didattici

Strutture Dati e Funzioni

- La libreria è la seguente (le funzioni mancanti sono lasciate per esercizio)

```
(defun somma-raz (r1 r2)
    (crea-razionale (+ (* (numer r1) (denom r2))
                       (* (numer r2) (denom r1)))
                    (* (denom r1) (denom r2))))

(defun molt-raz (r1 r2)
    (crea-razionale (* (numer r1) (numer r2))
                    (* (denom r1) (denom r2))))

(defun =-raz (r1 r2)
    (= (* (numer r1) (denom r2))
       (* (numer r2) (denom r1))))
```

Le cons-cells e la funzione CONS

- Una delle strutture dati più importanti in LISP è la cosiddetta **cons-cell**
- Una **cons-cell** è semplicemente una coppia di puntatori a due elementi

- Le **cons-cells** sono create dalla funzione **`cons`**, che si preoccupa di ***allocare la memoria*** necessaria al mantenimento della struttura
  - Pensate alla `cons` come una `new` in Java

`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

```
prompt> (defparameter c (cons 40 2))
c

prompt> (car c)
40

prompt> (cdr c)
2
```

Rappresentazione dei numeri razionali

- Data la primitiva cons, le funzioni **`crea-razionale`**, **`numer`** e **`denom`** diventano molto semplicemente

```
(defun crea-razionale (n d)
    (cons n d))

(defun numer (r) (car r))

(defun denom (r) (cdr r))
```

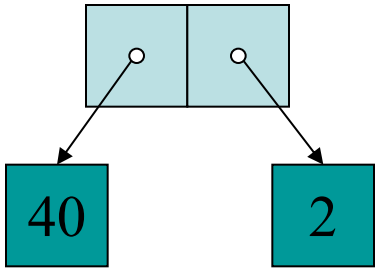
- Esempio

```
prompt> (denom (crea-razionale 42 7))
7
```

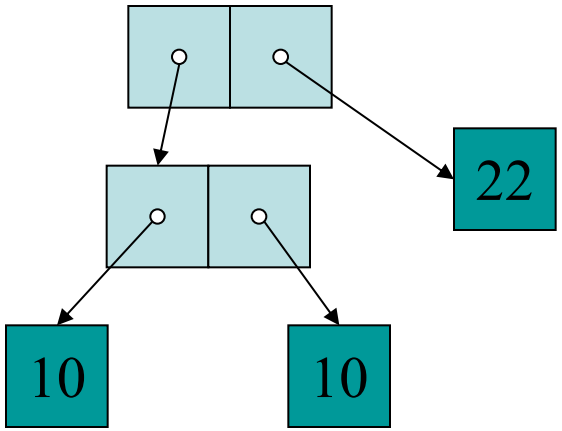
CONS

- La funzione **`cons`** genera in memoria dei grafi di puntatori arbitrariamente complessi
- Questi grafi vengono rappresentati in una tradizionale notazione detta *box-and-pointer*
- Esempio

```
(cons 40 2)
```



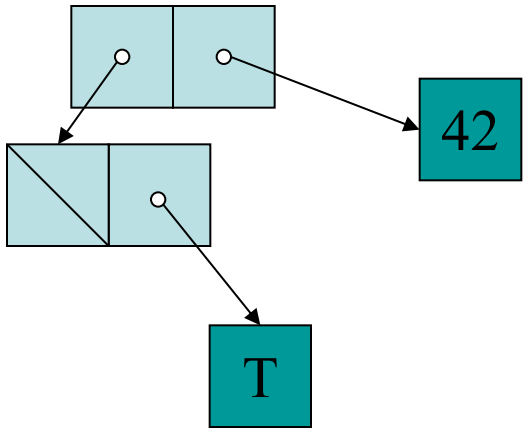

```
(cons (cons 10 10) 22)
```



CONS

- I valori `T`, `NIL` e stringhe sono valori quanto altri

```
(cons (cons NIL T) 42)
```



- Si noti la particolare rappresentazione di una cons-cell contenente un puntatore a **`NIL`**

CONS

- Come risponde il sistema Lisp quando si richiama la funzione **cons**?
- Esempio

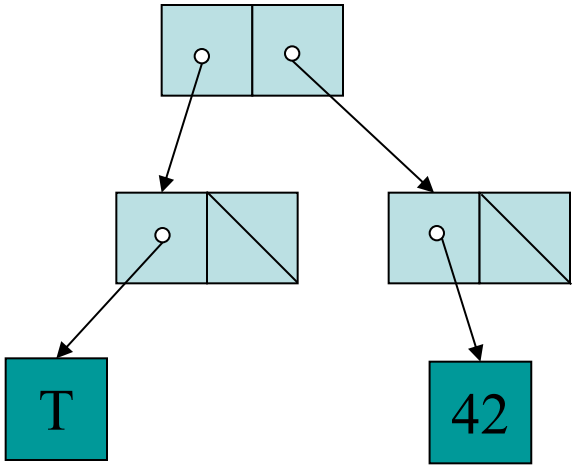
```
prompt> (cons NIL T)
(NIL . T)   ; Notazione "dotted-pair"
            ; ("coppia-puntata").  Gli spazi
            ; sono significativi.


prompt> (cons 4 2)
(4 . 2)


prompt> (cons "foo" 42)
("foo" . 42)
```

- La notazione "*dotted pair*" è l'unica "**irregolarità sintattica infissa**" in Lisp.
  - Come vedremo, questa notazione ha un'importante *abbreviazione*

CONS

- Quando si parla di liste in Lisp ci si riferisce a particolari configurazioni di cons-cells dove l'ultimo puntatore è **`NIL`**; la costante **`NIL`** è equivalente alla lista vuota **`()`**.

```
(cons (cons T NIL) (cons 42 NIL))
```



- Una lista Lisp corrispondente a questa configurazione di cons-cells viene rappresentata tipograficamente come

`((T) 42)` *che è equivalente a* **`((T . NIL) . (42 . NIL))`**

CONS

- … in altre parole una cons-cell con `NIL` come secondo elemento (ovvero come `cdr`) viene stampata senza il punto ed il `NIL`

- Esempio

```
prompt> (cons 42 nil)
(42)

prompt> (cons "foo bar" nil)
("foo bar")
```

CONS

- … inoltre, una cons-cell con una cons-cell come **secondo** elemento è stampata in modo “abbreviato”

- Esempio

```
prompt> (cons 42 (cons 123 666))
(42 123 . 666)      ; Equivalente a (42 . (123 . 666))

prompt> (cons 42 (cons "foo bar" nil))
(42 "foo bar")
;;; To`, una 'lista'.
```

Liste e la funzione `LIST`

- La funzione `cons` può quindi essere usata per rappresentare sequenze (**liste**) di oggetti
- Esempio

```
(defparameter L (cons 1 (cons 2 (cons 3 (cons 4 NIL)))))
```

genera in memoria una sequence di cons-cells tale che

```
prompt> (car (cdr (cdr L)))
3
```

- La costruzione è così utile da meritare un nome particolare

```
(defparameter L (list 1 2 3 4))
```

ottiene lo stesso effetto

Liste

- La funzione **`list`** accetta un numero variabile di argomenti
- Le liste vengono stampate in maniera succinta

```
prompt> (list -1 0 1 2 3)
(-1 0 1 2 3)
```

- ATTENZIONE
  L'espressione `(list 1 2 3)` e la lista `(1 2 3)` non vanno confuse: la prima è un'espressione a tutti gli effetti, la seconda è la rappresentazione tipografica di una struttura dati in memoria, che non ha un operatore nella posizione canonica
- Si noti anche che

```
prompt> (list)
NIL
```

Liste

- Ora che possiamo costruire varie liste abbiamo anche la possibilità di manipolarle
- Come si estrae l'*n*-esimo elemento da una lista?

```
(defun list-ref (n list)
    (if (<= n 0)
        (car list)
        (list-ref (- n 1) (cdr list))))
```

- Come calcoliamo la lunghezza di una lista?

```
(defun lunghezza (l)
    (if (null l)
        0
        (+ 1 (lunghezza (cdr l)))))
```

dove la funzione `null` ritorna il valore `T` se l'argomento passatole è il valore `NIL`

Elementi di una lista

- La funzione **`list-ref`** è definita nello standard Common Lisp con il nome **`nth`**

- La funzione `cdr` ritorna di fatto il “resto” di una lista e lo standard Common Lisp fornisce anche l’ovvio sinonimo: **`rest`**

- La funzione `car` ritorna il “primo” elemento di una lista; lo standard Common Lisp fornisce anche il sinonimo **`first`**

- Al fine di evitare le combinazioni standard

```
(car (cdr L))
(car (cdr (cdr L))
(car (cdr (cdr (cdr L))))
...
```

  lo standard Common Lisp ha in libreria le funzioni **`second`**, **`third`**, **`fourth`**, fino a **`tenth`**

Liste

- Come concateniamo due liste?

```
(defparameter pari (list 2 4 6 8 10))
(defparameter dispari (list 1 3 5 7))
```

- Vogliamo una funzione **appendi** che, dati **pari** e **dispari**, ritorni la lista

```
(2 4 6 8 10 1 3 5 7)
```

Liste

- La funzione **`appendi`** si può costruire così
  - Date due liste L1 ed L2
  - Se L1 è la lista vuota (ovvero è uguale a **`NIL`**) allora tanto vale ritornare L2 direttamente
  - Se L1 non è vuota allora possiamo prenderne il primo elemento ed attaccarlo in cima a….

….al risultato ottenuto dall' appendere il resto di L1 a L2

Liste

- La definizione di `appendi` è semplicemente

```
(defun appendi (l1 l2)
     (if (null l1)
         l2
         (cons (car l1) (appendi (cdr l1) l2))))
```

- La funzione è ricorsiva e presenta una tipica forma di ricorsione "strutturale" sul resto di una lista (detta anche "*cdr-recursion*" o "*rest-recursion*")

Sommario

- Chiamate di funzione, stacks, activation frames
- Operatore speciale **`LAMBDA`** e funzioni anonime
- Nomi (**simboli**) in Common Lisp
- Operatore speciale condizionale **`COND`**
- Funzioni ricorsive
  - Funzioni tail-ricorsive e loro relazione con il meccanismi di iterazione
- Strutture dati e **`CONS`** (e *cons-cells*)
  - Liste e **`LIST`**