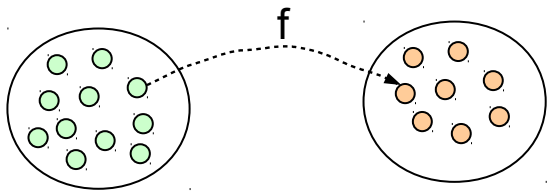
Programmazione Funzionale

- LP imperativi: apparenza simile $\rightarrow$ modello di progettazione = macchina fisica
- Famiglia dei LP imperativi = progressivo “miglioramento” del FORTRAN
- Obiezione: pesante aderenza dei LP alla macchina fisica =
  restrizione non necessaria al processo di sviluppo del software
- Possibilità di modelli alternativi di progettazione dei LP
- **Paradigma funzionale**: basato sulle funzioni matematiche $\rightarrow$ LP funzionali

Funzioni Matematiche

- **Funzione matematica** = mapping: *dominio* $\rightarrow$ *codominio* : specificato da { espressione, tabella }

- Funzione **applicabile** ad un elemento del dominio



- Valutazione della espressione di mapping: controllata da { **expr condizionali**, **ricorsione** }

$$\text{fact}(n) = \begin{cases} 1 & \text{se } n = 0 \\ n\ \text{fact}(n\text{-}1) & \text{se } n > 0 \end{cases}$$

- Funzione matematica: definisce un **valore** (non una sequenza di operazioni su variabili in memoria per produrre un valore)

- $\nexists$ { variabili, effetti collaterali } $\rightarrow$ valore della funzione dipende solo dai suoi argomenti

Funzioni Semplici

- Def di funzione: *nome* (*lista-di-parametri*) *expr-di-mapping*

```
cubo(x) = x*x*x
```

dominio = codominio = $\Re$

- Valutazione della expr di mapping: **x** rappresenta un (immutabile) elemento

- Applicazione della funzione: associazione *f elemento-del-dominio*

```
cubo(2.0) → 8.0
```

Forme Funzionali

- **Funzione di ordine superiore** 〈 parametri sono funzioni
  risultato è una funzione

Esempi:

1. **Composizione di funzioni**:

```
f(x) ≡ x+2
g(x) ≡ 3*x
```

⇨

```
h(x) ≡ f ∘ g ≡ f(g(x)) = (3*x)+2
```

2. **Costruzione**:

```
f(x) ≡ x*x
g(x) ≡ 2*x
h(x) ≡ x/2
```

⇨

```
[f, g, h](4) → (16, 8, 2)
```

3. **Applicazione universale**:

```
f(x) ≡ x*x

map(f,(2,3,4)) → (4,9,16)
```

Fondamenti dei LP Funzionali

- Obiettivo: massimizzazione delle similarità con funzioni matematiche
- Approccio al problem-solving: molto diverso da quello dei LP imperativi
- Programma = { def di funzioni } + { applicazioni di funzioni }
- Esecuzione = valutazione delle applicazioni delle funzioni
- Poichè $\nexists$ var assegnabili $\rightarrow$
  - impossibile controllare cicli (conteggio, condizionali) $\rightarrow$ ricorsione
  - $\nexists$ stato del programma $\rightarrow$ risultato dipende solo dagli argomenti

  ⇣

  trasparenza referenziale
- FPL fornisce
  - { funzioni primitive }
  - { forme funzionali } $\rightarrow$ costruzione di funzioni complesse partendo da primitive
  - operazione di applicazione delle funzioni
  - strutture dati
- Pb delle funzioni nei LP imperativi
  - restrizione sui tipi di valori restituiti $\rightarrow$ forme funzionali ridotte
  - effetti collaterali

Scheme

- 1958 < John Mc Carthy, Marvin Minsky > MIT AI Project → progettazione di un LP per la manipolazione di liste: **Lisp**

- Caratteristiche:
  - Solo due tipi di strutture dati < **atomi** < identificatori, sequenze di cifre ; **liste**: delimitate da parentesi

```
(A B C D)

(A (B C) D (E (F G)))
```

  - Paradigma funzionale → non necessario assegnamento
  - Omogeneità della rappresentazione di < dati, codice

```
(A B C D)
```

< lista di 4 elementi, applicazione di A a B, C, D

- Impatto: domina le applicazioni AI per 25 anni

- Discendenti del Lisp < **Common Lisp** { diversi tipi di strutture dati, scope statico e dinamico, packages ; **Scheme**

- Subset di Scheme → <u>senza</u> assegnamento → FPL Turing-completo quando { tipo intero, creazione di funzioni condizionali, ricorsione

Espressioni

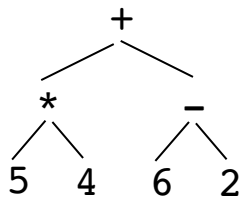
- Notazione prefissa con parentesi `(+ 2 3)` → possibilità di numero variabile di argomenti

```
(+)                    ; valutata 0
(+ 5)                  ; valutata 5
(+ 5 4 3 2 1)          ; valutata 15
(*)                    ; valutata 1
(* 5)                  ; valutata 5
(* 1 2 3 4 5)          ; valutata 120
```

- Liste per rappresentare { dati / funzioni } → primo elemento della lista { operatore / nome funzione }

- Espressioni complesse mediante liste innestate

`5*4+(6–2)` ⇨ `(+ (* 5 4) (- 6 2))`



- Valori globali `(define d 120)`

  ↳ cambia l'ambiente (unica nel nostro subset)

Valutazione delle Espressioni

- Governate da 3 regole:

1. Nomi sostituiti dai loro binding correnti

```
(define d 120)
   ...
d                          ; 120
(+ d 5)                    ; 125
```

2. Liste valutate come chiamate di funzioni scritte in forma prefissa

```
(+)                    ; invoca + senza argomenti
(+ 5)                  ; invoca + con un argomento
(+ 5 4 3 2 1)          ; invoca + con cinque argomenti
(+ (5 4 3 2 1))        ; errore: cerca di valutare 5 come una funzione
(d)                    ; errore: d non è una funzione
```

3. Costanti sono valutate come se stesse

```
5                ; valutato 5
#t               ; valutato true (predefinito)
#f               ; valutato false (predefinito)
```

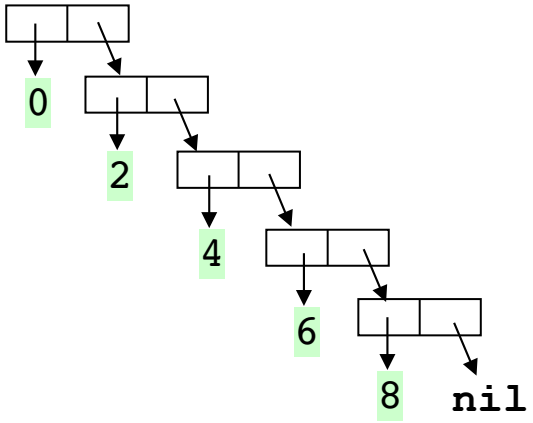
- Inibizione della valutazione di un nome

```
(define colori (quote (rosso giallo verde)))
(define colori '(rosso giallo verde))
```

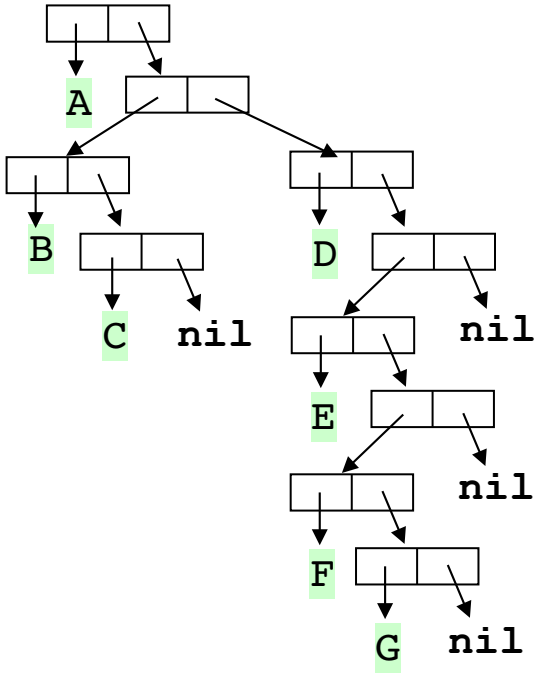
```
(define x d)                  ; definisce x col valore di d (120)
(define x 'f)                 ; definisce x come simbolo f
(define uncolore 'rosso)      ; definisce uncolore come rosso
(define uncolore rosso)       ; errore: rosso non definito
```

Liste

```
(define pari '(0 2 4 6 8))
(define dispari '(1 3 5 7 9))
```




```
(A (B C) D (E (F G)))
```



- Costruzione: (*cons elem lista*)

```
(cons 8 ())                          ; (8)
(cons 6 (cons 8 ()))                 ; (6 8)
(cons 4 (cons 6 (cons 8 ())))        ; (4 6 8)
```

Liste (ii)

- Lista = ***testa + coda***

```
(car pari)              ; 0
(cdr pari)              ; (2 4 6 8)
(car (cdr pari))        ; 2
(cadr pari)             ; 2
(cdr (cdr pari))        ; (4 6 8)
(cddr pari)             ; (4 6 8)
(car '(6 8))            ; 6
(cdr '(6 8))            ; (8)
(car '(8))              ; 8
(cdr '(8))              ; ()
```

- Costruzione di liste
  - `list` → crea la lista dei suoi argomenti
  - `append` → concatenazione dei due argomenti lista

```
(list 1 2 3 4)          ; (1 2 3 4)
(list '(1 2) '(3 4) 5)  ; ((1 2) (3 4) 5)
(list pari dispari)     ; ((0 2 4 6 8) (1 3 5 7 9))
(list 'pari 'dispari)   ; (pari dispari)
```

```
(append '(1 2) '(3 4))     ; (1 2 3 4)
(append pari dispari)      ; (0 2 4 6 8 1 3 5 7 9)
(append '(1 2) ())         ; (1 2)
(append '(1 2) (list 3))   ; (1 2 3)
```

Liste (iii)

- Test di emptyness: `null?`

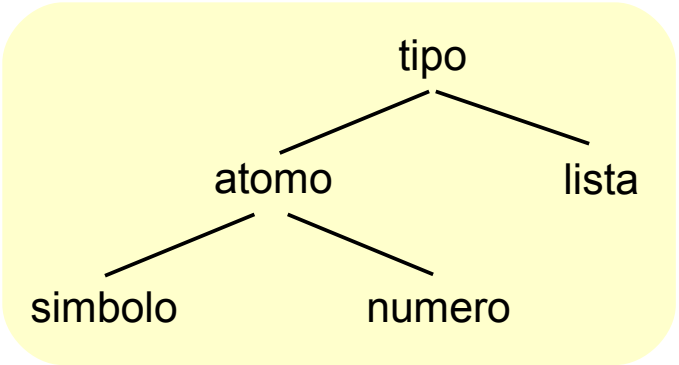
```
(null? '())              ; #t
(null? pari)             ; #f
(null? '(1 2 3))         ; #f
(null? 5)                ; #f
```

- Test di uguaglianza: `equal?`

```
(equal? 5 5)                      ; #t
(equal? 5 1)                      ; #f
(equal? '(1 2) '(1 2))            ; #t
(equal? 5 '(5))                   ; #f
(equal? '(1 2 3) '(1 (2 3)))      ; #f
(equal? '(1 2) '(2 1))            ; #f
(equal? () ())                    ; #t
```

- Test di tipo:

```
(list? pari)          ; #t
(symbol? 'pari)       ; #t
(number? 3)           ; #t
(atom? pari)          ; #f
```



Costrutti di Controllo

- Selezione a una via

```
(if test parte-then)
```

```
(if (< x 0)(+ x 1))
```

altrimenti, restituisce un nullo (**#void**)

- Selezione a due vie

```
(if test parte-then parte-else)
```

```
(if (< x y) x y)
```

Definizione di Funzioni

(**define** *nome* (**lambda** (*argomenti*) *corpo*)) ⟹ binding(*nome*, **λ-*astrazione***)

```
(define min (lambda (x y)(if (< x y) x y)))
```

⇕

```
(define (min x y)(if (< x y) x y))
```

```
(define (abs x)(if (< x 0) (- 0 x) x))
```

```
(define (fattoriale n)
   (if (< n 1) 1 (* n (fattoriale (- n 1)))))
)
```

Definizione di Funzioni (ii)

- Computazione iterativa → ricorsione

```
(define (sommatoria numeri)
   (if (null? numeri) 0
       (+ (car numeri) (sommatoria (cdr numeri))))
   )
)
```

piede della ricorsione

passo ricorsivo

- Lunghezza di una lista

```
(length '(1 2 3 4))          ; 4
(length '((1 2) 3 (4 5)))    ; 3
(length ())                  ; 0
(length 5)                   ; errore
```

```
(define (length lista)
   (if (null? lista) 0
       (+ 1 (length (cdr lista)))
   )
)
```

Definizione di Funzioni (iii)

- Appartenenza: `(member` *elemento lista*`)` $\begin{cases} \exists \rightarrow \text{resto della lista con testa } \textit{elemento} \\ \nexists \rightarrow \texttt{\#f} \end{cases}$

```
(member 4 pari)                    ; (4 6 8)
(member 1 pari)                    ; #f
(member 2 '((1 2) 3 (4 5)))        ; #f
(member '(3 4) '(1 2 (3 4) 5))     ; ((3 4) 5)
```

```
(define (member el lista)
   (if (null? lista) #f
       (if (equal? el (car lista)) lista
           (member el (cdr lista))
       )
   )
)
```

Definizione di Funzioni (iv)

- Sostituzione: `(subst` *elem1* *elem2* *lista*`)`

```
(subst 'x 2 '(1 2 3 2 1))            ; (1 x 3 x 1)
(subst 'x 2 '(1 (2 3) 2 1))          ; (1 (2 3) x 1)
(subst 'x 2 '(1 (2 3) (2)))          ; (1 (2 3) (2))
(subst 'x '(2 3) '(1 (2 3) 2 3))     ; (1 x 2 3)
(subst '(2 3) 'x '(x y x y))         ; ((2 3) y (2 3) y)
```

```
(define (subst x y lista)
   (if (null? lista) ()
       (if (equal? y (car lista))
           (cons x (subst x y (cdr lista)))
           (cons (car lista) (subst x y (cdr lista)))
       )
   )
)
```

Fattorizzazione di Sottoespressioni

- Identificazione di sottoexpr: `(let ((`id_1 $expr_1$`)(`id_2 $expr_2$`) ... (`id_n $expr_n$`)) `$corpo$`)`

  ⇨ valutazione della espressione una sola volta

```
(let ((x 2)(y 3))(+ x y))    ; valutata 5
```

```
(define (subst x y lista)
   (if (null? lista) ()
       (let ((testa (car lista))(coda (cdr lista)))
           (if (equal? y testa)
               (cons x (subst x y coda))
               (cons testa (subst x y coda))
           )
       )
   )
)
```

⇨ espressioni valutate tutte prima dei binding

- **`let*`** → binding (id_i, $expr_i$) realizzato prima della valutazione di $expr_{i+1}$

```
(define x 0)
(let ((x 2) (y x)) y)     ; 0
(let* ((x 2) (y x)) y)    ; 2
```

Regole di scope gerarchiche

Forme Funzionali

- Applicazione universale: `(mapcar` *funzione lista*`)`

```
(define (mapcar fun lista)
   (if (null? lista) ()
       (cons (fun (car lista)) (mapcar fun (cdr lista)))
   )
)
```

```
(define (quadrato x) (* x x))
(mapcar quadrato '(2 3 5 7 9))                ; (4 9 25 49 81)
(mapcar (lambda (x) (* x x)) '(2 3 5 7 9))    ; alternativa
```

Costruzione di Codice

- Uniformità strutturale di { dati / codice } ⟹ sfruttata per costruire programmi dinamicamente

- `eval` → valuta il suo argomento

```
(define (sommatoria numeri)
   (if (null? numeri) 0
       (+ (car numeri) (sommatoria (cdr numeri)))
   )
)
```

```
(define (sommatoria numeri)
   (eval (cons '+ numeri))
)
```

⟹

```
(define lista '(3 4 5))

(sommatoria lista)
```

⇩

```
(eval '(+ 3 4 5))
```

Costruzione di Codice (ii)

```
(eval 6)                                    ;   6
(eval (+ 1 2 3))                           ;   6
(eval '(+ 1 2 3))                          ;   6
(eval ''(+ 1 2 3))                         ;   (+ 1 2 3)
(eval '''(+ 1 2 3))                        ;   (quote(+ 1 2 3))
(eval (eval '''(+ 1 2 3)))                 ;   (+ 1 2 3)
(eval (eval (eval '''(+ 1 2 3))))          ;   6
(eval (eval ''(+ 1 2 3)))                  ;   6
(eval (eval (eval ''(+ 1 2 3))))           ;   6
```

```
(eval (length '(1 2 3)))                   ;   3
(eval '(length (1 2 3)))                   ;   errore
(eval '(length '(1 2 3)))                  ;   3
(eval ''(length (1 2 3)))                  ;   (length (1 2 3))
(eval '''(length (1 2 3)))                 ;   (quote (length (1 2 3)))
(eval ''(length '(1 2 3)))                 ;   (length (quote (1 2 3)))
(eval (eval ''(length (1 2 3))))           ;   errore
(eval (eval ''(length '(1 2 3))))          ;   3
```