Linguaggi di Programmazione

Quack

Linguaggi di Programmazione

Indice

Linguaggi di Programmazione

Prolog

Linguaggi di Programmazione

Cap 1. Introduzione al corso

Linguaggi di programmazione: Ci sono 3 tipi di paradigmi:

- **Imperativi**: procedurali
- **Logici**: rappresenta la conoscenza
- **Funzionali**: funzione

Paradigma imperativo: Il paradigma imperativo è basato su Von-Neumann: memoria (componente passiva) e processore (componente attiva). Il concetto di variabile è un'astrazione di cella di memoria.
Ci sono linguaggi a livelli di astrazione diversi, cioè sono a distanze diverse da Von Neumann: Asm, Pascal, C.
Stile prescrittivo perché danno istruzioni. Programma: Algoritmi + Strutture Dati. Sono stati creati altri paradigmi per: aumentare astrazione da Von Neumann e aumentare la semplicità del codice

Paradigma funzionale e logico: I due nuovi paradigmi hanno in comune alcuni aspetti: Linguaggi ad altissimo livello. Manipolazione simbolica e non numerica. Basati su concetti matematici. Programma e Strutture dati non nettamente separate. Non si usano variabili. Linguaggi dichiarativi.

Paradigma Logico: Definire un linguaggio logico significa definire come il programmatore può esprimere la conoscenza e quale tipo di controllo si può utilizzare nel processo di deduzione. Programma = conoscenza + controllo.
Problema: rappresentazione conoscenza limitata.

Paradigma funzionale: Associazione tra due insiemi, tra dominio e codominio. Concetto primitivo: funzione.
La definizione di funzione specifica dominio, codominio, regola di associazione. Dopodiché la si applica e la si valuta
Il concetto di variabile è quello di costante matematica. Programma = Composizione di Funzioni + Ricorsione

Linguaggi di Programmazione

Cap 1. Introduzione al corso

Confronto prescrittivo e dichiarativo: Devo ordinare una lista:

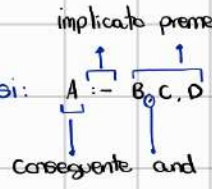
Prescrittivo: Programmatore specifica la sequenza di istruzioni che servono a generare la sequenza di permutazioni di L. Controllo se L è vuota, se sì dai la lista vuota. Altrimenti calcola una permutazione L1 di L e controllo se è ordinata, se sì dai L1, altrimenti calcolo un'altra permutazione di L ecc...

Dichiarativo: Genera da solo le possibili permutazioni di L secondo un processo di deduzione matematica.

- Il risultato dell'ordinamento di una lista vuota è la lista vuota.
- Il risultato dell'ordinamento di una lista L è L1 se L1 è ordinata ed è la permutazione di L.

Linguaggio Prolog: Paradigma Logico

- Asserzioni incondizionate (fatti): lavora(ugo, samsung)
- Asserzioni condizionate (regole): lavora(x,y)
- Interrogazione (query): :-lavora(ugo,y)



Linguaggio Lisp: Paradigma funzionale

- Non c'è assegnazione.
- Unica struttura: lista
- Si esprime in funzioni
- Computazione ricorsiva

Ambienti run-time: Un ambiente run-time deve mantenere lo stato di computazione e gestire memoria fisica e virtuale. Questo avviene tramite lo stack e l'heap.

Stack e activation frame: Lo stack è la memoria per la gestione delle chiamate. Quando una di queste si attiva si crea un activation frame, che contiene le informazioni per gestire la chiamata: il return address Variabili locali al metodo e valori di ritorno, static link, che punta all'ambiente globale, dynamic link, che punta all'activation frame argomenti, parametri passati.

L'Heap e il garbage collector: L'Heap serve per gestire le strutture dinamiche. Il garbage collector si occupa di pulire dalla memoria le allocazioni a strutture non più utili.

Linguaggi di Programmazione

Cap 1. Introduzione al corso

Tabella di confronto

	IMPERATIVO	LOGICO	FUNZIONALE
Esempio	Assembly, Pascal, C	Prolog	Lisp
OOP	✓	✓	✓
Basato	Von Neumann	Conoscenza	Funzione
Livello	Basso	Altissimo	Altissimo
Stile	Prescrittivo	Dichiarativo	Dichiarativo
Programma	Algoritmi + Strutture Dati	Conoscenze + Controllo	Composizione Funzione + Ricorsione
Variabile	Cella di memoria	Costante matematica	Costante matematica
Manipolazione	Numerica	Simbolica	Simbolica
Svantaggi	Poca astrazione	Rappr. limitata conoscenza	

Linguaggi di Programmazione

Cap 2: Introduzione alla logica e ragionamento

Dobbiamo capire come un ragionamento può essere formalizzato con un n° di passi connessi da regole a partire da premesse per arrivare a delle conclusioni.

Regole di inferenza: le regole d'inferenza fanno parte del calcolo naturale o di Gentzen.

Un insieme di regole di inferenza costituisce la base di un calcolo logico. Lo scopo è quello di manipolare in modo sintattico.

Due logiche ne fanno uso: logica proposizionale e logica del primo ordine

Logica Proposizionale: Si occupa delle conclusioni che traiamo da proposizioni. È definita sintatticamente da un insieme P di proposizioni.

L'interpretazione o valutazione è quando associo una funzione di verità V a ogni proposizione P.

Connettivi logici: congiunzione $\wedge$, disgiunzione $\vee$, negazione $\neg$, implicazione $\rightarrow$. Un calcolo logico crea un processo di generazione che si chiama dimostrazione.

Regole di inferenza:

- Modus ponens $\frac{A \rightarrow B, A}{B}$
- Modus tollens $\frac{A \rightarrow B, \neg B}{\neg A}$
- Terzo escluso $\frac{p \vee \neg p}{\text{vero}}$
- Eliminazione $\wedge$ $\frac{p_1, p_2, \ldots, p_k}{p_i}$
- Introduzione $\wedge$ $\frac{p_1, p_2, \ldots, p_k}{p_1 \wedge p_2 \ldots \wedge p_k}$
- Introduzione $\vee$ $\frac{p}{p \vee q}$
- Unit Resolution (Prolog): $\frac{\neg a \rightarrow b,\ b \rightarrow c}{\neg a \rightarrow c}$ $\qquad \frac{a \vee b,\ \neg b \vee c}{a \vee c}$

Principio di risoluzione: Il principio di risoluzione (base interprete prolog) è una regola d'inferenza generalizzata.

Opera su FBF in forma congiunta. Si basa su l'estensione di nozione di rimozione dell'implicazione sulla base del principio di contraddizione.

Solitamente si usa con dimostrazioni per assurdo.

Dimostrazione per assurdo: Voglio dimostrare p. Assumo che $\neg p$ sia vero. Cerco di arrivare a contraddire p. Quindi $\neg p$.

Linguaggi di Programmazione
Cap. 3: Teoria Prolog

Base formale: Calcolo dei predicati del 1° ordine ma in forma di clausole di Horn. Dimostrazione attraverso unit resolution. Le clausole di Horn son clausole che hanno al più un letterale positivo.

Ogni fbf può essere riscritta in: congiunta: $\bigwedge_i (\bigvee_j L_{ij})$ o disgiunta: $\bigvee_i (\bigwedge_j L_{ij})$

Prolog contiene solo fatti e regole. Dà informazioni sul sistema.

Sintassi: Le espressioni sono chiamate termini e possono essere atomi, variabili o comp. di termini.

Un atomo si riconosce con caratteri alfanumerici con 1° carattere minuscolo, o racchiuso tra apici, o numero o stringa.

Una variabile inizia con lettera maiuscola o con _.

I funtori sono simboli di predicati o di funzione applicato ai suoi argomenti

Interrogazioni (query o goals): Quando Prolog controlla se è vera la query, la pone a false e se la trova nella base di conoscenza diventa true.

Unificazione variabili: unificare significa assegnare ad una variabile.

Es: libro(Kawaleri, Prolog). libro(Yoshimoto, Kitchen) ?-libro(x, Prolog) ⟶ {x / Kawaleri}

Liste prolog: [elemento1, elemento2]

lista vuota [] es: [a, b, c] testa: a, coda: [b, c]

testa: primo elemento [a] testa: a, coda: []

separatore: si può usare | per separare l'inizio e la coda di una lista.

Dimostrazione: Una dimostrazione avviene se riesco ad ottenere la clausola vuota (.)

Risoluzione ad input lineare: Prolog dimostra un goal attraverso una sequenza di passi di risoluzione. Questa avviene sempre tra l'ultimo goal derivato in ciascun passo e una clausola di programma. (SLD)

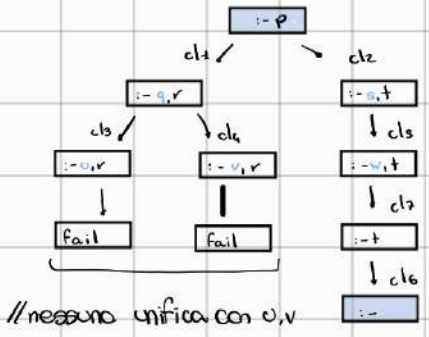
Linguaggi di Programmazione
Cap. 3: Teoria Prolog

Attraversamento: La regola di attraversamento dei termini sono

- **left-most:** si usa a sostituire la prima clausola a sx
- **right-most:** si sostituisce la clausola più a dx.

Dimostrazione con albero (left most)

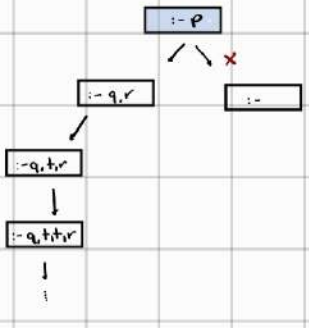
- goal :- p
- cl: cl_1: p:-q,r , cl_2: p:-s,t , cl_3: q:-u , cl_4: q:-v , cl_5: s:-w , cl_6: t , cl_7: w



Regole di calcolo: influiscono sulla struttura di un albero per ampiezza e profondità. Non influiscono su correttezza e completezza

Qualunque sia R, il n° di cammini di successo è lo stesso in tutti gli alberi SLD costruiti per $P \cup \{G_0\}$

Fallimento infinito: Se a sx continuo a generare ad incremento e a dx ho :-, ho un fallimento infinito!



Pile: Prolog utilizza due pile durante l'esecuzione:

- **Esecuzione**, contiene i record di attivazione delle "procedure" (sostituzioni per unificazione delle regole)
- **Backtracking**, contiene i "punti di scelta"

Linguaggi di Programmazione
Cap. 3: Teoria Prolog

Backtracking: Il Prolog usa un predicato speciale, chiamato **cut** (!) per controllare le scelte.

Il cut ci indica che se dobbiamo dimostrare qualcosa e non ci siamo riusciti con le clausole a sx, l'unica altra scelta disponibile è quella a dx del cut. Se anche questa fallisse, non posso effettuare altre eventuali sostituzioni e dunque non posso risolvere!

Albero di derivazione con cut

Clausole: cl1) g:-a.

cl2) g:-s.

cl3) a:- p, !, b.

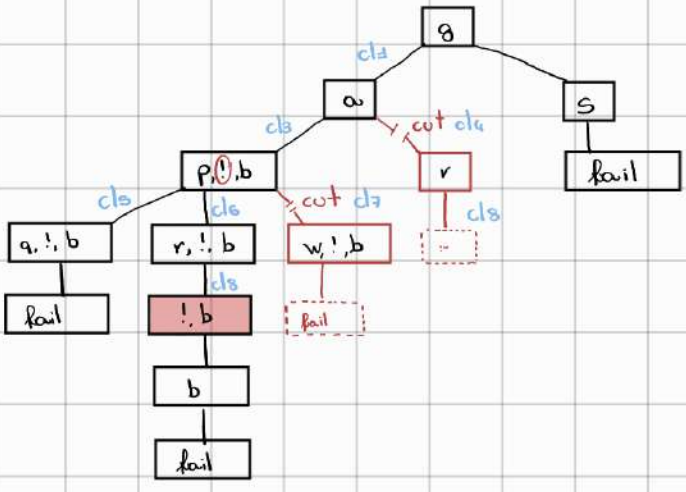
cl4) a:-r.

cl5) p:-q.

cl6) p:-r.

cl7) p:-w.

cl8) r.



Tipi di cut: Si distinguono due tipi di cut:

- **Green Cuts:** utili per esprimere determinismo. (+ efficiente)
- **Red Cuts:** sono indesiderabili in quanto modificano la semantica del programma.

Linguaggi di Programmazione

Cap. 4: Altri concetti su Prolog

Ispezione di termini: 3 predicati che modificano termini:

- `functor(term, F, arity)` true se term è un termine con F = nome e Arity = n° elementi
- `arg(N, Term, Arg)` vero se l'N-esimo di Term è Arg.
- `Term =.. L`, chiamato **univ**, vero quando L è una lista il cui primo elem. è funtore di Term e i restanti suoi argomenti.

Programmazione di ordine superiore: È anche chiamato del secondo ordine. Ci permette di ottenere come risultato l'insieme di tutte le istanze che soddisfano una query. Prolog mette a disposizione una serie di predicati su insieme. I più importanti sono 3:

- `findall(Template, Goal, Set)`: vero se Set contiene tutte le istanze di Template che soddisfano Goal.
- `bagof(Template, Goal, Bag)`: vero se Bag contiene tutte le alternative di Template. Può avere ∃ con "^"
- `setof(Template, Goal, Set)`: come bagof ma set non ha duplicati.

Questi predicati vanno usati con attenzione, in quanto modificano dinamicamente lo stato del programma.

- `listing`: stampa la base di conoscenza.
- `assert`: inserisce elementi nella base di conoscenza. Si possono asserire fatti e regole.
- `retract`: rimuove elementi nella base di conoscenza
- `asserta`: Inserisce all'inizio
- `assertz`: Inserisce alla fine.

Linguaggi di Programmazione

Cap. 4: Altri concetti su Prolog

I/O: I predicati primitivi sono read e write, a cui si aggiungono quelli per gestione degli streams.

read e write agiscono su termini prolog.

Automi: riconoscere nondeterministicamente dei linguaggi regolari:

```
accept([I|I_s],S):-
    delta_s(S,I,N).   // S,N stati , I passo
    accept(I_s,N).    // Se S va in N tramite I , chiamo ricorsivamente accept senza primo elemento.
accept([],Q):-final(q).
```

Per decidere se una certa sequenza di simboli è riconosciuta dall'automa costruiamo il predicato:

```
recognize(Input):- initial(s), accept(Input,s)
```

Automi a pila: riconoscere nondeterministicamente dei linguaggi context-free

```
accept([I|I_s],Q,S) :-    // Input, Stato, Pila
    delta_s(Q,I,S,Q_1,S_1),
    accept(I_s,Q_1,S_1)
accept([],Q,[]):-final(q).
```

Per decidere se una certa sequenza di simboli è riconosciuta dall'automa a pila costruiamo il predicato:

```
recognize(Input):- initial(s), accept(Input,s,[])
```

Il predicato call è il più semplice meta-interprete, ovvero degli interpreti più complicati se accettiamo di rappresentare con sintassi leggermente diversa.

Linguaggi di Programmazione

Lisp

Linguaggi di Programmazione

Cap 1: Introduzione al paradigma funzionale e a Lisp

Paradigma funzionale: I programmi computano combinando valori trasformati da **chiamate a funzioni**.

Effetto collaterale: caratteristica negativa dei linguaggi imperativi: passaggio di parametri per **riferimento**. Ci sono modifiche dello stato di memoria di un programma (es.: dim array variabili, puntatori).

Problemi: distinguere effetti collaterali desiderati e non, leggibilità del programma

è difficile verificare la correttezza di un programma

Esempio: $w := x + f(x, y) + z$. La funzione f può modificare x e y se passati per indirizzo e z se globale.

$v := x + z + f(x, y) + f(x, y) + x + z$. In questo caso abbiamo due valori diversi per $x+z$.

Trasparenza referenziale: il significato del **tutto** si può determinare dal significato delle **parti**.

Questo permette la sostituzione di funzioni se denotano gli stessi valori. Nel paradigma imperativo non era assicurato che $f(x) + g(x) = g(x) + f(x)$ o che $f(x) + f(x) = 2f(x)$.

Programma funzionale: un programma è una funzione matematica. Una funzione può essere una funzione **primitiva** o una **composizione** di funzioni

Struttura linguaggio: Oltre alle funzioni e la composizione di funzioni, avremo **metodi** per costruire **astrazioni** per poter fare riferimento a gruppi di espressioni per nome e **operatori speciali**.

Linguaggi di Programmazione

Cap 1: Introduzione al paradigma funzionale e a Lisp

3 fasi paradigma funzionale:

- **Definizione**: Specificare dominio, codominio e regola di associazione.
- **Applicazione**: Applicare la funzione agli elementi del dominio.
- **Valutazione**: Restituzione dell'elemento del codominio associato.

Ricorsione: spesso le espressioni sono organizzate ricorsivamente e controllate da operatori speciali.

Funzioni di ordine superiore: Nei linguaggi funzionali è permesso l'uso di funzioni che hanno come argomento altre funzioni.

Autovalutanti: In LISP, le stringhe e i numeri sono autovalutanti, avero il loro valore è la rappresentazione stessa.

Esempio: > 42 → 42 > "ciao" → "ciao"

Notazione prefissa: Tutte le funzioni in LISP hanno il nome che le denota come 1° argomento delle parentesi. Essendo anche gli operatori aritmetici delle funzioni, anche loro seguono lo standard.

Esempio: >(+ 40 2) → 42 (+2 (* 2 10) 20) → 42

Defparameter: Operatore speciale che permette di definire delle variabili

Esempio: > (defparameter quarantadue 42) → quarantadue. >quarantadue → 42

Defun: Operatore speciale che permette di definire funzioni.

Esempio: (defun quadrato(x) (* x x)) → quadrato >(quadrato 4) → 16

Linguaggi di Programmazione

Cap 2: Introduzione ad operatori e liste

Lambda expression: l'operatore lambda denota una funzione anonima.

Esempio: `((lambda (x) (+ x 42))`

Cond: operatore speciale quando dobbiamo controllare più casi.

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

If: operatore speciale per quando abbiamo una situazione if-else.

Esempio:
```
(defun valore-assoluto (x)
    (if (> x 0) x (-x))     // if x > 0 is x else -x.
```

Booleani: In LISP sono presenti operatori booleani come and, or, not

Funzioni ricorsive: Prendiamo due funzioni ricorsive apparentemente simili: Fattoriale e Fibonacci.

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

```
(defun fibonacci (n)
    (if (or (= n 0) (= n 1)) 1
        (+ (fibonacci (- n 2)) (fibonacci (- n 1)))))
```

La principale differenza tra le due ricorsività è che Fattoriale può essere riscritto con un accumulatore e quindi "simulando" iteratività.

Linguaggi di Programmazione

Cap 2: Introduzione ad operatori e liste

Funzioni tail ricorsive: Quando lo compilatore può ottimizzare la chiamata ricorsiva con una JUMP senza creare un nuovo record di attivazione

Esempio: Fattoriale come ciclo con accumulatore tail-ricorsivo.

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

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

Nota bene: le funzioni con due chiamate ricorsive non sono tail-ricorsivi.

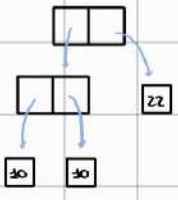
Cons-cell: Struttura dati formata da una coppia di puntatori: car e cdr.

Esempio: `(defparameter c (cons 40 2))` → c `(car c)` → 40 `(cdr c)` → 2.

`(defun razionale (n d) (cons n d))` → razionale `(defun numeratore (r) (car r))` → numeratore.

`(defun denominatore (r) (cdr r)` → denominatore `(numeratore (razionale 42 7))` → 42

Grafico: `(cons (cons 10 10) 22)`



Dotted pair: la risposta del compilatore a `(cons 4 2)` è `(40 . 2)` ed è una irregolarità in quanto è in notazione infissa

Casi: `(cons 42 NIL)` → 42 `(cons 42 (cons 123 2))` → `(42 123 . 2)`

Liste: Una lista in LISP è una serie di cons cell dove l'ultimo puntatore è NIL.

Abbiamo l'operatore speciale list

Esempio: `(defparameter L (list 1 2 3 4))` → L.

Linguaggi di Programmazione

Cap 2: Introduzione ad operatori e liste

Operazione su liste: Possiamo manipolare le liste.

Esempio: estrarre n-esimo elemento di una lista: `(defun estrai (n list)`

```
(if (= n 0) (car list)      //1° elem.
    (estrai (- n 1) (cdr list)))  //scorro
```

Estrazione elemento da lista: Operatore speciale nth.

Rest: Quando si estrae un elemento da una lista, il resto viene incluso nel predicato rest.

Second-Tenth: Quando lavoriamo con car e cdr, possiamo riferirci al 2°, 3°, ecc..., 10° elemento col nome.

Esempio: `(second m) = (car (cdr (m))`, `(third m) = (car (cdr (cdr m)))`.

Cdr recursion: Quando, lavorando sulle liste, abbiamo ricorsione sulla coda.

Append: Funzione che concatena due liste.

```
(defun append (l1 l2)
  (if ((null l1) l2)
      (cons (car l1) (append (cdr l) l2)))).
```

Linguaggi di Programmazione

Cap 3: Valutazioni. Ricorsioni Doppie

Quote: Operatore speciale che permette di assegnare un valore ad un elemento in LISP

Esempio: > A → errore > 'A → A (1 2 3) → errore '(1 2 3) → (1 2 3)

Autovalutanti: Elementi che non necessitano di quote poiché il loro valore è la rappresentazione tipografica stessa. È il caso di numeri e stringhe.

Symbolic Expressions: Anche dette Sexp's, sono l'unione di atomi (numeri, simboli e stringhe) e di cons-cell (liste). In LISP i programmi e sexp's sono equivalenti.

Atom: Predicato che controlla se un argomento è un atomo o no.

Esempio: (atom 'L) → T, (atom (cons 42 42)) → NIL

Eval: Predicato che applica la funzione al suo argomento definendo i legami di quest'ultimo in cascata.

Esempio: definita la somma di una lista come

(defun sum(l) (if (null l) 0 (+ (first l) (sum (rest l)))))

la funzione (eval (sum '(3 4 5))) produrrà in output:

```
> (eval (sum '(3 4 5)))
    => (+ 3 (sum '(4 5)))
        => (+ 4 (sum (5)))
            => (+ 5 (sum ()))
            <= (+ 5 0)
        <= (+ 4 5)
    <= (+ 3 9)
<= 12
```

Linguaggi di Programmazione

Cap 3: Valutazioni. Ricorsioni Doppie

Ricorsione doppia: Quando bisogna effettuare ricorsione anche sulla testa. Sono tutte nella forma: 1. Base, 2. Ipotesi ricorsiva e 3. Passo. Vediamo alcuni esempi

count-atoms conta gli atomi di una sexp. (count-atoms '((a b c) 1 (xyz d))) → 6

```
(defun count-atoms (x)
    (cond ((null x) 0)
          ((atom x) 1)
          (T (+ (count-atoms (first x)) (count-atoms (rest (x)))))))
```

prof: Profondità massima di una Sexp's (prof '(a (b ((c)) (d)))) → 4.

```
(defun prof (x)
    (cond ((atom x) 0)
          ((null x) 1)
          (t (max (+1 (prof (first x))) (prof (rest x))))))
```

flatten: appiattire una lista (flatten '(((a (b (c d)) e) f)) → (a b c d e f)

```
(defun flatten (x)
    (cond((null x) x)
         ((atom x) (list x))
         (T (append (flatten (first x)) (flatten (rest x))))))
```

mirror: specchiare una sexp's. (mirror '(a (b (c d)) e) f)) → (f (e ((d c) b) a)

```
(defun mirror (x)
    (if ((atom x) x)
        (append (mirror (rest x)) (append (first x))))).
```

Linguaggi di Programmazione

Cap 4: Funzioni su Liste

eql: Predicato per controllare che simboli e interi siano uguali.

equal: Predicato che applica eql a tutti gli atomi di una lista.

mapcar: Astrazione "applica la funzione f a tutti gli elementi della lista L e ritorna una lista di valori".

Es: vogliamo una funzione che moltiplica tutti gli atomi di un certo numero:

```
(defun scala-lista-10 (lista)
    (mapcar 'scala-10 lista)) dove (defun scala-10 (x) (* x 10)).
```

lambda: Con l'operatore lambda possiamo creare delle funzioni senza dare loro un nome.

Esempio: L'esempio di prima diventa
```
(defun scala-lista (lista fattore)
    (mapcar (lambda (e) (* e fattore)) lista)).
```

let: Predicato che permette di inserire dei valori intermedi da riutilizzare.

Esempio: la funzione $f(x,y) = x(1+xy)^2 + y(1-y) + (1-y)(1+xy)$ è riscrivibile come:

```
(defun f (x y)
    (let ((a (+ 1 (* x y)))
          (b (- 1 y)))
        (+ (* x (quadrato a))
           (* y b)
           (* a b)))).
```

(a = $1+xy$, b = $1-y$)

Funzioni di ordine superiore: Funzioni che prendono come argomenti altre funzioni, come mapcar. Vediamo anche:

- compose
- filter
- accumula

Linguaggi di Programmazione

Cap 4: Funzioni su Liste

compose: corrisponde alla nozione matematica di funzione.

Semantica: `(defun compose (f g)`
`(lambda (x)`
`(funcall f (funcall g (x))))`.

Funcall: serve per chiamare una funzione con un certo argomento.

filter: rimuove dalla lista gli elementi che non soddisfano il predicato.

Semantica:
```
(defun filter (predicato lista)
    (cond ((null lista) nil)
          ((funcall predicato (car lista))   //applico predicato al 1° elemento
           (cons (car lista)                 //se soddisfa lo metto nella lista
                 (filter predicato (cdr lista))))
          (T (filter predicato (cdr lista)))))
```

accumula: simbolo di funzione da distribuire a tutti gli elementi se ho un punto di partenza.

Semantica:
```
(defun accumula (f iniziale lista)
    (if (null lista) iniziale
        (funcall f (car lista) (accumula f iniziale (cdr lista)))))
```

Esempi:
- `(accumula '+ 0 '(1 2 3))` → 6
- `(accumula '* 1 '(1 2 3 4))` → 24
- `(accumula 'cons NIL '(1 2 3))` → (1 2 3)

Linguaggi di Programmazione

Cap 4: Funzioni su Liste

iota: permette di creare una lista decrescente a partire da un certo valore.

Semantica: `(defun iota (n) (if (= n 0) nil (cons n (iota (- n 1)))))`.

Esempio: Con iota possiamo ridefinire fattoriale come

```
(defun fattoriale (n)
  if (zerop n) 1 (accumula '* 1 (iota n))))
```

lambda list: lista con argomenti variabili:

`(defun foo (a b c &rest l) (append l (list a b c)))`

× param. fissi indicatore parametri variabili parametro contenente la lista degli argomenti

subseq: funzione che prende in input una lambda-list come la funzione substring:

Semantica: `(defun subseq (a &optional l) (cons a l))`

Esempi: `(subseq "qwerty" 2)` → erty `(subseq "qwerty" 1 4)` → "wer"

Keywords: si possono definire delle funzioni che utilizzano delle Keywords, ovvero parametri associati ai loro nomi.

find: funzione che restituisce l'elemento cercato all'interno della lista se presente.

Esempi: `(find 2 '(1 2 3 4 5 6))` → 2

`(find 2 '(1 2 3 4 5 6) :start 3)` → NIL

`(find 2 '(1 2 3 4 5 6) :start 1 :end 3)` → 2

parametri a chiave: con la sintassi `(defun nome (&key x y) list(x y))` possiamo passare parametri che diventano Keyword da poter usare al momento della chiamata.

Linguaggi di Programmazione

Cap 5: I/O in Lisp

Read: serve per leggere e valutare.

Esempio: (third (read)) (foo 41 (42) 43 44) ⟶ (42).

Print: stampa un oggetto LISP rispettandone la sintassi

Esempio: (print "Hello world") ⟶ "Hello world".

Format: simile alla fprintf C/C++. Le direttive sono introdotte dal carattere ~:

- ~D stampa numeri interi
- ~% va a capo
- ~S stampa un oggetto LISP secondo la sua sintassi.
- ~A stampa un oggetto LISP secondo la sintassi esteticamente piacevole.

Esempio: (format t "~S è una stringa! ~%" "foo") ⟶ "foo è una stringa!

Streams: Ci sono 3 tipi di streams in LISP:

- standard input
- standard output
- standard error

Le funzioni read print e format accettano un n° variabile di parametri, tra cui streams, nello specifico output per format e print e input per read.

Manipolazione file: Con la macro (with-open-file (<var> <file> :direction :I/O) <codice>)

REPL: Read-eval-print loop, ovvero il ciclo principale dell'Interprete LISP, nello specifico della sua command-line.

Linguaggi di Programmazione

C, C++

Linguaggi di Programmazione

Cap 1: Introduzione a C

Compilazione ed esecuzione: per compilare si usa `gcc nomefile.c`, l'eseguibile sarà `a.out`, che viene chiamato direttamente senza altri comandi. Con opzione `-o` si può dare un nome specifico

Elementi: tutti gli elementi sono denotati da nomi, ovvero stringhe alfanumeriche + simbolo "_".

Tipi di dato: I dati primitivi sono:

- int e unsigned int
- char e unsigned char
- float
- bool
- puntatori e riferimenti

Il C ha due tipi aggregati:

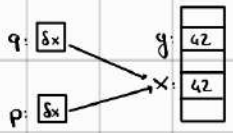
- Array di int, char e float
- Strutture, contengono tipi diversi

Le stringhe sono degli array di char terminati con un carattere nullo '\0'.

Tutti i tipi di dati vanno dichiarati.

Puntatori: Dato un qualunque tipo T, il tipo associato T* è il puntatore a T, ovvero contiene l'indirizzo di memoria di un oggetto di tipo T. L'indirizzo si denota con &x.

es:
```
int x = 42;
int* p = &x;
int* q = p; // q punta a ciò che punta q
int y = *q; // deferenziazione
```



I puntatori possono essere usati negli array: `int a[] = {1,2,3,4}, int* p1 = v;` // primo elemento: 1

`int* p2 = v[2];` // punta al 3 elemento: 3

`int* p3 = v[4];` errore!

Linguaggi di Programmazione

Cap 1: Introduzione a C

Statement: return <espressione>, goto <label>, in C++ anche try-catch.

Passaggio di parametri: con i puntatori si passano i riferimenti, es: void swap(int* a, int* b), dal main si chiama con swap(&a, &b).

Compilatore: I file si dividono in Headers file .h e file di implementazione .c

Chiamare il compilatore fa si che viene chiamato anche il preprocessore e sul linker.

Si può scegliere se chiamare solo 1 tra preprocessore e compilatore con -E, il secondo con -c.

Produzione di eseguibili

file .h e file.c → pre-processore → file c → compilatore → file.o → Linker → a.out

Produzione di librerie

file .h e file.c → pre-processore → file c e file i → compilatore → file.o → Linker → Lib.so/dll

Preprocessore: trasforma testo. Opera su 3 direttive: inclusione di testo, definizione di macro, condizionali.

es: #define, #include

Compilazione separata: ogni programma dovrebbe essere modularizzato, ovvero con compilazione separata, per evitare errori con il linker.

Libreria: è un file in un particolare formato che viene manipolato dal linker, estensione .so; è una collezione di file oggetto con un indice che permette al linker di caricare il codice ad un "entry point".

Linguaggi di Programmazione

Cap 1: Introduzione a C

Memoria dinamica: non esiste il garbage collector. Viene allocata con **malloc** e deallocata con **free** su C, mentre con **new** e **delete** su C++

Malloc: restituisce un puntatore di tipo void* ad una zona di memoria nell'heap. Se non vi è memoria disponibile, malloc restituisce un puntatore nullo

Modificatori di dichiarazioni: abbiamo 2 modificatori.

- **extern**: definizione non locale (utile per interfacce)
- **static**: dichiarazione fissata nella memoria globale e non visibile al di fuori di essa

Const: permette di dichiarare costanti.

Stdio.h: abbiamo tre streams fondamentali:

- stdin / std::cin input
- stdout / std::cout output
- stderr / std::cerr error

Output: le funzioni più semplici sono:

- int **fputc**(int c, FILE* ostream): scrive carattere c su ostream
- int **fputs**(const char* s, FILE* ostream) scrive la stringa s su ostream
- int **fprintf**(FILE* ostream, const char* format, ...) scrive format su ostream dopo averlo interpretato sulla base degli argomenti.

In C++ si usa l'operatore << dopo cout.

Linguaggi di Programmazione

Cap 1: Introduzione a C

Input: più complicato dell'output. Le funzioni principali sono:

- `int fgetc(FILE* istream)`: legge e restituisce un carattere da istream.
- `char* fgets(char* s, int n, FILE* istream)` legge fino a n caratteri da istream nella stringa s e restituisce. se vi è newline si ferma.
- `int fscanf` simmetrica a `fprintf`.

In C++ si usa l'operatore `>>`.

File I/O in C: `fopen` apre file, `fclose` per chiuderlo, `remove` lo elimina.

- `FILE* fopen(const char* filename, const char* mode)`. Mode può essere:
  - `"r"`: apre in lettura
  - `"w"`: apre azzerando o crea nuovo file in scrittura
  - `"a"`: append.
- `int fclose(FILE* stream)` ritorna 0 se va a buon fine, `EOF` altrimenti.

In C++ si usano `ifstream` e `ofstream`