

LINGUAGGI DI PROGRAMMAZIONE

 Brock

INTRODUZIONE ALLA LOGICA

PROCESSO DI DEMOSTRAZIONE D \equiv S \vdash F in cui D è una prova, S è l'insieme delle affermazioni note e

F la frase da provare

- F è una conseguenza di S, e S è una sequenza di passi in cui $P_n = F$ e ogni P_i può essere ottenuto mediante una regola di inferenza.

↳ un insieme di regole = base del calcolo logico

LOGICA PROPOZIONALE si occupa delle conclusioni che possiamo trarre da un insieme di proposizioni

- è quindi definita da un insieme P di proposizioni
- all'insieme P è associata una funzione di verità V: P $\rightarrow \{\text{Vero, Falso}\}$

FBF Formule ben formate, l'insieme di tutte le formule formate dagli elementi di P e dalle loro combinazioni

LETTERALI Formule atomiche (e negazioni) in P

DIMOSTRAZIONE processo di generazione di nuove FBF

REGOLE DI INFERNZA Serie di regole ben formate che ci permettono di ottenere nuove formule a partire da una serie di assiomi. Hanno forma: $\frac{F_1, \dots, F_n}{R}$ [nome regola]

- **MODUS PONENS** ci permette di aggiungere le conclusioni di un'implicazione all'insieme di FBF.
È nella forma $\frac{P \Rightarrow q, P}{q}$ [modus ponens]

e.g. $p \Rightarrow q$: se piove, allora la strada è bagnata

p: piove

q: (allora) La strada è bagnata

- **MODUS TOLLENS** ci permette di aggiungere la premessa negata di una regola al nostro insieme di FBF

È nella forma $\frac{P \Rightarrow q, \neg q}{\neg P}$

e.g. $p \Rightarrow q$: se piove, allora la strada è bagnata

$\neg q$: la strada non è bagnata

$\neg p$: (allora) non piove

- **ELIMINAZIONE/INTRODUZIONE DI E** ci permette di aggiungere alle FBF i singoli componenti di una congiunzione.

Sono nelle forme: $\frac{P_1 P_2 \dots P_n}{P_i}$ [eliminazione], $\frac{P_1 P_2 \dots P_n}{P_1 P_2 \dots P_n}$ [introduzione]

e.g. Piove e la strada è bagnata

(segue che) piove

- **INTRODUZIONE DI O** ci permette di aggiungere i singoli componenti di una formula complessa (addizione)

È nella forma $\frac{P}{P \vee q}$ [introduzione v]

e.g. Piove

Piove o c'è vita su Marte

- esistono altre regole di inferenza secondarie:

$\frac{P \vee \neg P}{\text{verò}}$ [terzo escluso]

$\frac{\neg \neg P}{P}$ [eliminazione \neg] $\frac{P \wedge \neg P}{q}$ [contraddizione] \rightarrow si può trarre
qualsiasi conseguenza

PRINCIPIO DI RISOLUZIONE

regola di inferenza generalizzata

- opera su FBF trasformate in **Forma normale congiunta**
- ogni congiunto è detto **clausola**

RISOLUZIONE UNITARIA quando una delle due clausole da risolvere è un letterale.

È nella forma $\frac{\neg p, q_1 \vee \dots \vee q_n \vee p}{q_1 \vee \dots \vee q_n}$

e.g. (da) non piove, piove o c'è il sole

(segue che) c'è il sole

DIMOSTRAZIONE PER ASSURDO

dato un insieme di FBF, dimostrare che una certa proposizione p è vera.

Procediamo assumendo $\neg p$ come vera. Se, combinandola con le FBF ottengo una contraddizione, allora concludo che p è vera.

e.g. $\text{FBB} = \{p \Rightarrow q, p, \neg w, e, r\}$, assumere $\neg q$.

q è vera poiché $\text{FBB} \cup \{\neg p\}$: $p \Rightarrow q \equiv \neg p \vee q$, combinato p produce q .

quindi abbiamo $q \wedge \neg q$ (ottengo \perp).

ASSIOMI sono conoscenze pregresse

- alcune proposizioni sono sempre vere, ovvero **tautologie**.

e.g. come costruire delle prove da una base di conoscenza? Consideriamo:

Se l'unicorno è mitico, allora è immortale, ma se non è mitico allora è mortale.

Se è mortale o immortale, allora è cornuto.

L'unicorno è magico se è cornuto.

1 Identificare le proposizioni: $UM = \text{Uni. è mitico}$

$$UI = \text{Uni. è immortale}$$

$$UMag = \text{uni. è magico}$$

$$UC = \text{Uni. è cornuto}$$

2 Trascrivere le proposizioni: $UM \Rightarrow UI$

$$\begin{aligned} & UM \Rightarrow UI \\ & \neg UI \vee UI \\ & \neg UI \vee UI \Rightarrow UC \\ & UC \Rightarrow UMag \end{aligned} \quad \left. \begin{array}{c} \\ \\ \end{array} \right\} S$$

3 Rispondere alla domanda: l'unicorno è cornuto? Ovvero $S \vdash UC$?

$$P_1: \neg UI \vee UI \Rightarrow UC \quad \text{da } S$$

$$P_2: \neg UI \vee UI \quad \text{terzo escluso}$$

$$P_3: UC \quad \text{da } P_1, P_2 \text{ e modus ponens}$$

Rispondere a $S \vdash UMag$?

$$P_1: \neg UI \vee UI \Rightarrow UC \quad \text{da } S$$

$$P_2: \neg UI \vee UI \quad \text{terzo escluso}$$

$$P_3: UC \quad \text{da } P_1, P_2 \text{ e modus ponens}$$

$$P_4: UC \Rightarrow UMag \quad \text{da } S$$

$$P_5: UMag \quad \text{da } P_3, P_4 \text{ e modus ponens}$$

TAUTOLOGIA Sono le regole seguenti:

De Morgan: $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$

$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$$

Implicazione: $A \Rightarrow B \Leftrightarrow \neg A \vee B$

SINTASSI E SEMANTICA un calcolo logico fornisce una manipolazione sintattica, mentre la semantica di un insieme di formule dipende dalla funzione di valutazione V .

- introduzione di \models , conseguenza logica
- **TEO DI COMPLETEZZA E VALIDITÀ**: $S \vdash f$ se e solo se $S \models f$
- **MODELLO** interpretazione di V che rende vere tutte le formule in S .

LOGICA DEL PRIMO ORDINE la log. proposizionale non consente di rappresentare insiemi di elementi in maniera concisa (e.g. tutti gli uomini sono mortali.)

Introduciamo quindi il primo ordine, costituito da **termini** costruiti a partire da:

- V simboli di variabili
- C simboli di costante
- R simboli di relazione (arità)
- F simboli di funzione
- Connettivi Logici e simboli di quantificazione \forall (universale) e \exists (esistenziale)

REGOLE DEL PRIMO ORDINE:

- $\frac{\forall x. T(\dots, x, \dots), c \in C}{T(\dots, c, \dots)}$ [eliminazione \forall]
- $\frac{T(\dots, c, \dots), c \in C}{\exists x. T(\dots, x, \dots)}$ [introduzione \exists]
- $\exists x. \neg T(\dots) \equiv \neg \forall x. T(\dots)$
- $\forall x. \neg T(\dots) \equiv \neg \exists. T(\dots)$

e.g. Socrate è un uomo.

Tutti gli uomini sono mortali.

Allora socrate è mortale.

Definiamo: $C = \{ \text{Socrate}, \text{Platone}, \text{Aristotele} \}$

$R = \{ \text{uomo}, \text{mortale} \}$

Tradurre le asserzioni principali: $\forall x. (\text{uomo}(x) \Rightarrow \text{mortale}(x))$
 $\text{uomo}(\text{Socrate})$

e le conclusioni da trarre: $\text{mortale}(\text{Socrate})$

Segue quindi: $\frac{(\forall x. \text{uomo}(x) \Rightarrow \text{mortale}(x)), \text{Socrate} \in C}{\text{uomo}(\text{Socrate}) \Rightarrow \text{mortale}(\text{Socrate})}$ [eliminazione \forall]

$\frac{\text{uomo}(\text{Socrate}), \text{uomo}(\text{Socrate}) \Rightarrow \text{mortale}(\text{Socrate})}{\text{mortale}(\text{Socrate})}$ [eliminazione \Rightarrow]

INTRODUZIONE A PROLOG

PROGRAMMAZIONE LOGICA utilizzare La logica matematica come base dei linguaggi di programmazione
Il programma è visto come insieme di formule

PROLOG stile dichiarativo, è usato per determinare se una certa affermazione è vera o no e, se è vera, quali vincoli sui valori attribuibili alle variabili hanno generato la risposta

FORMA NORMALE ogni FBF può essere riscritta in forma normale a clausole

Vi sono due forme:

- **Forma normale congiunta** è una congiunzione di disgiunzioni $\wedge (\bigvee L_i)$
- **Forma normale disgiunta** è una disgiunzione di congiunzioni $\vee (\bigwedge L_i)$

FORMA NORMALE CONGIUNTIVA considerando una wff in CNF, scartando il simbolo di congiunzione rimangono solo le clausole disgiuntive:

$$(p(x) \vee q(x,y) \vee \neg t(z)) \wedge (p(w) \vee \neg s(u) \vee \neg r(v)) \rightsquigarrow p(x) \vee q(x,y) \vee \neg t(z) \\ p(w) \vee \neg s(u) \vee \neg r(v)$$

Inoltre, possiamo riscrivere come $t(z) \Rightarrow p(x) \vee q(x,y)$
 $s(u) \wedge r(v) \Rightarrow p(w)$

Ovvero come un insieme (congiunzione) di implicazioni.

CLAUSOLE DI HORN clausole che hanno al più un solo letterale positivo

- non tutte le fbf possono essere trasformate in clausole di Horn.
- i programmi prolog sono collezioni di clausole di Horn.

COSA CONTIENE PROLOG non contiene istruzioni, ma solo:

- **fatti**: asserzioni vere nel contesto che stiamo descrivendo
- **regole**: ci danno gli strumenti per dedurre nuovi fatti da quelli esistenti

Un programma ci dà informazioni su un sistema ed è chiamato **base di conoscenza**.

Inoltre non si esegue, ma si interroga (**query**).

Le espressioni sono chiamate **termini**.

ATOMO è una sequenza di caratteri alfanumerici, che inizia con un carattere minuscolo

- '...', numero, stringa

VARIABILE è una sequenza alfanumerica che inizia con un carattere maiuscolo o con il carattere _, che sono dette di **indifferenza** o **anonime**.

- vengono istanziate con il procedere del programma e vengono mostrate nella risposta

TERMINI COMPOSTI una composizione di termini consiste in:

- un **funtore** simbolo di funzione o predicato
- una sequenza di termini racchiusi tra parentesi tonde e separati da virgole, chiamati **argomenti** del funtore

Non ci deve mai essere uno spazio tra i due.

FATTO o predicato, consiste in un nome di predicato che inizia con una lettera minuscola e zero o più argomenti. Devono essere terminati da un punto.

REGOLE per esprimere che un certo fatto dipendente da un insieme di altri fatti.

Una regola è formata da una testa e di un corpo, collegati dall'operatore :-.

- La testa corrisponde al conseguente
- il corpo all'antecedente

Corrispondono alle clausole di Horn.

RELAZIONI può essere definita da più regole aventi lo stesso predicato come conclusione.

e.g. gentore(x,y) :- padre(x,y)

gentore(x,y) :- madre(x,y)

Può anche essere definita ricorsivamente.

UNIFICAZIONE dati due termini, la procedura di unificazione crea un insieme di sostituzioni delle variabili. Questo insieme permette di rendere uguali i due termini.

- l'operatore di unificazione è $'=$

MOST GENERAL UNIFIER insieme di sostituzioni, ogni sostituzione è indicata come una sequenza ordinata di coppie chiavi/valori

LISTE si definisce lista una collezione di elementi racchiusi tra parentesi quadre e separati da virgolette.

Può essere vista in due parti:

- la **testa** è il primo elemento della lista
- la **coda** tutto il resto ed è sempre una lista.

OPERATORE | distingue inizio e coda di una lista, come $[X|Xs]$.

La lista vuota è trattata in maniera speciale.

CONSULT predicato che consulta un file da valutare.

Può essere usato anche per inserire direttamente fatti o regole con il termine consult (user).

- si ricarica con **reconsult**.

ESECUZIONE DI UN PROGRAMMA considerando almeno un Goal Go da provare, si deve dimostrare che da $Pv\{Go\}$ è possibile derivare la clausola vuota, ovvero effettua una dimostrazione per assurdo mediante applicazione del principio di Risoluzione.

RISOLUZIONE AD INPUT LINEARE dimostra la veridicità o meno di un'interrogazione eseguendo una sequenza di passi di risoluzione, la risoluzione avviene sempre fra l'ultimo goal derivato in ciascun passo e una clausola di programma.

Il risultato finale può essere: successo viene generata la clausola vuota

insuccesso finito se per n finito, Gn non è uguale a :-
insuccesso infinito se è sempre possibile derivare nuovi risoltori

ALBERI DI RISOLUZIONE SLD dato un programma logico P, un goal G₀ e una regola di calcolo R, un albero SLD per P u {G₀} via R è definito sulla base del processo di prova:

- la radice è il goal
- ciascun nodo è un goal

La regola R è variabile:

- Left-most scelta del sottoalbero più a sinistra
- Right-most scelta del sottoalbero più a destra
- Scelta a caso
- Scelta del "migliore"

Il Prolog adotta una strategia di attraversamento dell'albero SLD in profondità (**depth first**) con **backtracking**.

Ad ogni ramo dell'albero corrisponde una derivazione SLD di successo. Il numero di cammini di successo è la stessa qualsiasi regola di calcolo si scelga, bensì influenza il numero di cammini di fallimento.

PROLOG E PROGRAMMAZIONE LOGICA

MODELLO DI ESECUZIONE

- **Goal** può essere visto come chiamata ad una procedura
- **Regola** definizione di una procedura (intestazione :- corpo)

Le clausole nel database vengono considerate da sinistra verso destra e dall'alto al basso.

Se un goal fallisce, allora prolog sceglie un'alternativa (la sequenza può essere manipolata con !, il **cut**).

L'esecuzione si basa su due stack:

- di **esecuzione** che contiene i record di attivazione delle procedure, ovvero le sostituzioni per l'unificazione delle varie regole
- di **backtracking** che contiene l'insieme dei "punti di scelta"; ad ogni fase della valutazione, contiene dei puntatori alle scelte aperte nelle fasi precedenti della dimostrazione

CUT è un predicato di controllo che permette di interrompere il meccanismo di backtracking, lo fa eliminando tutti i punti di scelta creati da quando si è entrati nel predicato nel quale appare il cut. Non modifica l'albero di derivazione, ma solo il suo attraversamento.

TIPI DI CUT si possono distinguere due tipi:

- **green cuts** per esprimere determinismo e rendere più efficiente il programma
- **red cuts** per scopi di efficienza: omettono alcune condizioni esplicite e modificano la semantica del programma senza cuts (sono tendenzialmente indesiderabili)

DETERMINISMO un programma Prolog è deterministico quando una sola delle clausole serve / si vorrebbe servisse per provare un goal.

ALTRI ELEMENTI DI PROLOG

PREDICATI META-LOGICI servono per stabilire quale è l'input e quale è l'output in predici non invertibili (e.g. convertitore Celsius / Fahrenheit e viceversa), che non hanno quindi la tipica invertibilità di varie queries.

Questo si dà all'uso di vari predici aritmetici ($>$, $<$, $=$, \leq , is , ...) che sacrificano la semantica.

Trattano le variabili come oggetti:

- var(X): vero se X è una variabile logica
 - nonvar(X): opposto di var(X)
- } il loro uso permette di stabilire quale clausola utilizzare

ISPEZIONE DI TERMINI dato un termine Term:

- atomic(Term): vero se Term è numero o costante
- compound(Term): vero se non atomic(Term)
- functor(Term, F, Arity): vero se Term è un termine, con Arity argomenti, il cui funtore è F
- arg(N, Term, Arg): vero se l'n-esimo argomento di Term è Arg
- Term =.. L chiamato univ., è vero quando L è una lista il cui primo elemento è il funtore di Term ed i rimanenti elementi sono i suoi argomenti

PROGRAMMAZIONE DI ORDINE SUPERIORE ci sono delle richieste che non sono formulabili direttamente al primo ordine, Prolog mette a disposizione dell'utente una serie di **predicati su insiemi** che estendono il modello computazionale:

- findall(Template, Goal, Set) vero se set contiene tutte le istanze di template che soddisfano goal.
 - bagof(Template, Goal, Set) vero se bag contiene tutte le alternative di template che soddisfano goal.
- Con Var[^]G si definiscono le variabili che non vanno considerate libere.
- setof(Template, Goal, Set) bagof, senza duplicati
 - call(G) :- G

MANIPOLAZIONE DELLA BASE DI DATI esistono dei predici che manipolano direttamente la base di dati:

- listing: stampa la base di dati
 - assert(X): asserisce X
 - retract(X): inverso di assert (rimuove solo una asserzione, usando una variabile si possono eliminare più asserzioni).
- assertz: accoda
→ asserta: appende

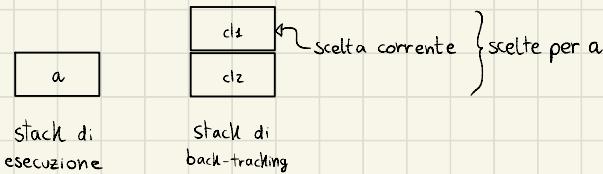
La manipolazione è utile per la tecnica di **memoizzazione** o **caching**.

EXPLAINED: MODELLO DI ESECUZIONE

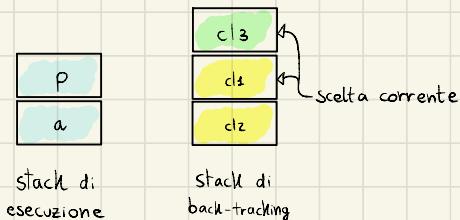
Iniziamo considerando:

- (c1) $a :- p, b$
- (c2) $a :- pc$
- (c3) $p.$

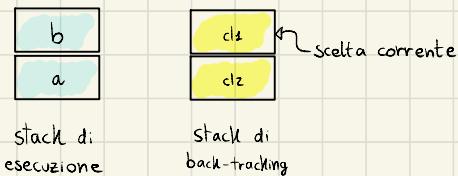
Seguendo il programma, si comincia dalla query $? - a$.



Si mette p in cima allo stack



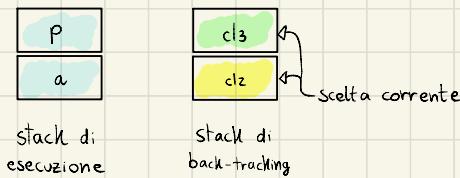
La valutazione di p ha successo, e si inserisce b nello stack:



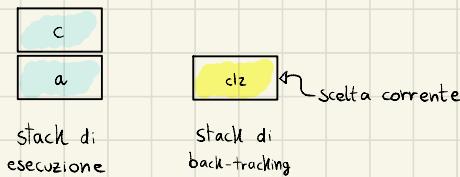
Ma la valutazione fallisce, quindi si attiva il backtracking, e si passa a considerare la seconda clausola.

Segue ...

Si mette p in cima allo stack:



La valutazione ha successo, si inserisce c:



Ma fallisce, e così via...

Alla fine di tutto, il backtracking è "tornate indietro" ad uno stato dal quale si può riprendere la computazione.

(o almeno credo, sono troppo stanco per approfondire).

EXPLAINED: ALBERO DI DERIVAZIONE

Considerare: (c1) $p :- q, r$.

(c2) $p :- s, t$

(c3) $q :- u$.

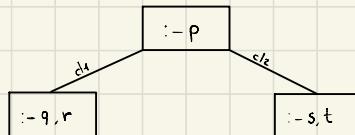
(c4) $q :- v$.

(c5) $s :- w$.

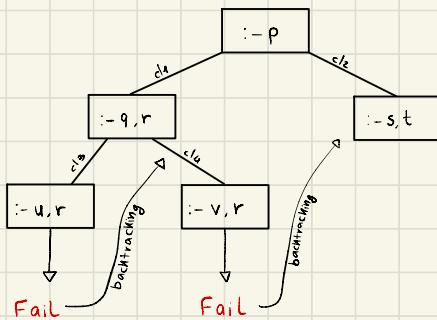
(c6) t .

(c7) w .

Prolog ragiona dall'alto verso il basso e da sinistra verso destra, quindi il primo goal è dimostrare $: - p$, e per farlo possiamo usare le due clausole c1 e c2, quindi l'albero inizialmente è:

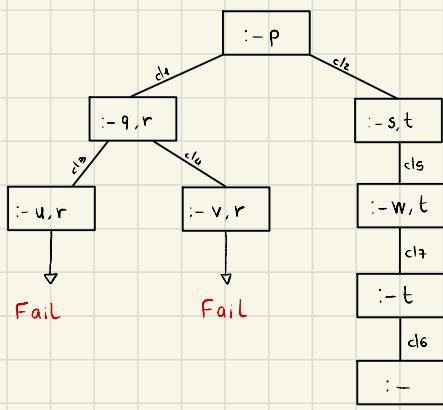


Continuiamo dall'alto verso il basso, sviluppiamo prima c1 (derivazione left-most, può anche essere richiesta la right-most, basta usare il ramo più a destra invece di procedere a sinistra).



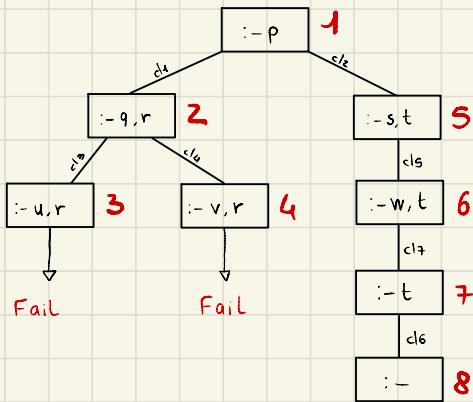
Entrambi i rami falliscono, perché non c'è modo di dimostrare $: - r$. Adesso l'algoritmo di backtracking si attiva, e si procede a verificare l'altro ramo, c12.

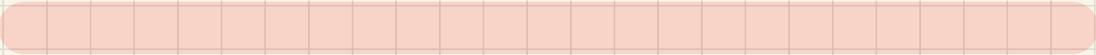
Segue...



L'albero raggiunge un ramo di successo quando il ramo termina con una foglia con la clausola vuota `:`, il che vuol dire che tutte le clausole sono verificate con successo.

Per renderlo più chiaro, ecco l'albero con i nodi numerati in ordine di attraversamento:





Secondo
Compitino

INTRODUZIONE AL C / C++

NOMI i vari elementi del linguaggio sono denotati da nomi / identificatori.

- Sono stringhe di lettere, numeri ed il carattere _
- alcuni nomi sono riservati

TIPI FONDAMENTALI

int interi da -2^{31} a $2^{31}-1$ } unsigned

char caratteri da -128 a 127

float numero floating

bool booleani

puntatori / riferimenti

array tipo aggregato

strutture aggregazione di campi di tipo diverso

void nullo

enum [nome] {val1, ..., Valn}

VARIABILI il c deve manipolare valori che vanno associati a nomi

- vengono introdotti con le dichiarazioni

ARRAY sono da considerarsi come un blocco di memoria fisica

STRINGHE sono degli array terminati con un carattere nullo '\0'

STRUTTURE sono aggregazioni di tipi diversi

- i campi si estraggono con la notazione punteggiata

PUNTATORE da un qualsiasi tipo T, il tipo associato T* è il tipo puntatore a T.

- T* contiene l'indirizzo in memoria di un oggetto di tipo T

- **DEREFERENZIAZIONE** con & si ottiene l'indirizzo

int x = 42;

int* p = &x;

int* q = p; // assegna puntatore a un altro

int y = *q; // dereferenza : y vale 42

PUNTATORI E ARRAY il nome di un array non è altro che un puntatore al primo elemento dell'array

OPERAZIONI SUI PUNTATORI dato p , $p+n$ va' n puntatori oltre il primo.

FUNZIONI un programma è costituito da un insieme di funzioni

- main è la funzione principale
- tutti i parametri sono passati per valore, tranne gli array
- il passaggio per riferimento si effettua con &, il tipo **reference**

PUNTATORI E STRUCT l'accesso a uno struct può essere fatto in modo equivalente con:

- $(^x).name$
- $x \rightarrow name$

INTRODUZIONE AL C / C++ 2^a parte

Il codice sorgente è di solito distribuito su un insieme di files e directories

- headers file
- file di implementazione

PRODUZIONE DI ESEGUITIBILI

file.c → pre-processore → file.i → compilatore → file.o → linker → a.out
file.h → file.c → file.exe

DIRETTIVE

#include inclusione

#define definizione

- condizionali #ifdef/#ifndef ... #else ... #endif

PREPROCESSORE introduce le costanti simboliche e processa i file.h

COMPILAZIONE SEPARATA ogni programma è modularizzato in più file, il compilatore agisce su un solo file, produce un file con dei riferimenti irrisolti a codice non direttamente disponibile.

- Il linker risolve i riferimenti.
- Ogni oggetto può essere definito una volta sola } un modo corretto, è usare gli header file solo per le dichiarazioni
- serve consistenza nelle dichiarazioni }

LIBRERIA è un file in un particolare formato che può essere manipolato dal linker.

- può essere statica o dinamica
- è essenzialmente una collezione di file oggetto con un indice associato

MEMORIA DINAMICA si è obbligati a gestire manualmente la memoria dinamica (**heap**).

- non esiste la garbage collector
- malloc alloca nello heap
- free dealloca dallo heap

e.g. int *p = (int *) malloc (10 * sizeof(int))

MALLOC restituisce un puntatore di tipo void* ad una zona di memoria nello heap.

- puntatore nullo se non vi è memoria disponibile
- le dimensioni del blocco dipende dal parametro passato

IN C++ si usano gli operatori **new** e **delete**.

INTRODUZIONE AL C / C++ 3^a parte

MODIFICATORI DI DICHIARAZIONE

- **extern** la dichiarazione ha una definizione non locale, ovvero la definizione dell'oggetto dichiarato si trova più in là nel file od in un altro file (e.g. file di interfaccia)
- **static** viene fissata nello spazio di memoria globale e non è visibile al di fuori del file in cui essa appare (e.g. definizioni globali in un file)
 - Variabili static mantengono il loro valore

CONSTANTI

Non sono modificabili

- const deve saper distinguere puntatore e puntato, sempre con la certezza che il suo valore non può essere modificato ma il puntato sì

```
char* pr;
char s[] = "Ford Prefect";

const char* pc = sr;
pc[3] = 'X';           // Errore!
pc = pr;

char *const cp = sr;
cp[3] = 'Y';           // Errore!
cp = pr;               // Errore.

const char *const cpc = sr;
cpc[3] = 'Z';           // Errore.
cpc = pr;               // Errore.
```

- Si può usare anche per non permettere la modifica degli array passati come parametri alle funzioni
- è utile in alternativa al #define

FILE

Sono legate al file system e usano degli streams associati ai file

- gli stream sono stdin, stdout, stderr e stdio.h
- in C++ esistono le classi ifstream e ofstream
 - l'operatore << scrive un valore su un output stream
 - l'operatore >> legge un valore da un stream

TYPEDEF

Sono abbreviazioni di strutture complesse

CODE CON PRIORITÀ ha diverse applicazioni, dai sistemi operativi alla ricerca su grafi, isolamento frodi, guida driverless...

- si può implementare con un array ordinato o meno, gravando sulla complessità delle operazioni elementari insert e extract
- con gli heap si può implementare in tempo logaritmico

INTRODUZIONE AL PARADIGMA FUNZIONALE

IDEA alla base è che un programma è costitutivo dalla combinazione di varie funzioni, primitive e composte, mantenendo la trasparenza referenziale propria della matematica

- richiede quindi la manipolazione di funzioni

FUNZIONE regola per associare gli elementi di un insieme a quelli di un altro insieme

- può essere applicata a un elemento del dominio, e produce un elemento del codominio
- gli argomenti sono variabili matematiche, non ha senso modificarla
- sono oggetti di prima classe, quindi:
 - possono essere parte di una struttura dati
 - possono essere costruite durante l'esecuzione di un programma

FUNZIONI DI ORDINE SUPERIORE funzioni che prendono altre funzioni come argomenti e che possono restituirlne come valore

RICORSIONE non esistono costrutti come while e for, si basa tutto sulla ricorsione combinata con gli operatori speciali

ESPRESSIONI COMPLESSE sono costruite mediante composizioni di funzioni $F = G \circ H = G(H(x))$

- sono spesso organizzate ricorsivamente
- non è possibile produrre effetti collaterali

LISP famiglia di linguaggi funzionali, con costrutti imperativi di "comodità"

- è un linguaggio interpretato
- le espressioni più semplici sono numeri e stringhe

NOTAZIONE PREFISSA le operazioni elementari possono essere scritte come $\otimes(x, y)$, e.g. $+(1, 5), -(4, 2), \dots$

- in lisp, la notazione diventa $(f\ x_1\ x_2 \dots x_n)$ con parentesi e spazi tra gli argomenti

ORDINE DI VALUTAZIONE procede da sinistra verso destra, applicando f agli argomenti

VARIABILI si definiscono con (defparameter name value).

DEFINIZIONI DI FUNZIONI

si definiscono con l'operatore speciale **defun**.

- (**defun** <nome della funzione> (<lista parametri formali>) (<corpo della funzione>))
- associa il corpo della funzione al nome nell'ambiente globale

VALUTAZIONE DI FUNZIONI

avviene mediante la costruzione di **active frames**.

- i parametri formali vengono associati ai valori (tutto si passa per valore)
- ad ogni sottoespressione del corpo si sostituisce il valore che essa denota
- il valore/i restituito dal corpo della funzione è il valore del corpo della funzione
 - quando il valore finale viene ritornato, l'act.frame viene rimosso

LAMBDA

Lisp ammette la creazione a runtime di funzioni, grazie alle strutture **closures**.

- è l'operatore che ci permette la costruzione di funzioni anonime
- (**(lambda** (x₁...x_n)<e>)) dove le x_i sono i parametri formali ed <e> un'espressione
- possiamo creare tutte le funzioni che vogliamo senza assegnare loro un nome

OPERATORI CONDIZIONALI

definizione matematica per casi, in cui il valore finale della funzione dipende dalla verità o meno della condizione preliminare

- ogni coppia viene considerata in ordine: se ci è T, ritorna e_i, altrimenti va avanti nella computazione
- Se non vi sono più coppie allora produce NIL.

OPERATORI BOOLEANI

- **and** [T]
- **or** [NIL]
- **not**

CONS-CELL una coppia di puntatori a due elementi:

- **cons** allora la memoria necessaria al mantenimento della struttura
- i due puntatori sono **car** e **cdr**, che sono anche funzioni
- non è altro che una dotted-pair, unica irregolarità sintattica infissa
- può essere usata per rappresentare delle **liste**

LIST accettano un numero variabile di argomenti, costruendo una lista

- **LIST** ≡ (cons 1 (cons 2 (cons 3)))
- **list-ref** definita come **nth**
- **rest** ritorna il resto della lista

QUOTE (quote <e>) l'espressione non viene valutata e ritorna letteralmente.

- abbreviato '
- numeri e stringhe sono **autovalutanti** (valore = rappresentazione)
- Le liste invece sono valutabili
- i programmi e i dati sono la stessa cosa

ATOMI simboli, numeri, stringhe...

- i non-atomi sono le cons-cell
- (atom <a>) verifica se a è un atomo

SYMBOLIC EXPRESSION Sono numeri, simboli e stringhe...

- **eval** applica la funzione al suo argomento, definendo i legami di N in cascata e costruendo una espressione che alla fine verrà valutata, a partire dalla sotto-espressione più annidata

FUNZIONI DI UGUAGLIANZA Sono vari:

- **eql** controlla l'uguaglianza di simboli e numeri interi
- **equal** come **eql**, ma è in grado di controllare se due liste sono uguali

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, date due funzioni, ritorna una nuova funzione $= f(g(x))$

FILTER rimuove gli elementi della lista che non soddisfano il predicato

ACCUMULA applica una funzione ad un elemento di una lista ed al risultato (ricorsivo) dell'applicazione di accumula al resto della lista.

LAMBDA LIST &rest <lista di argomenti>

KEYWORD associate parametri a nomi, con la sintassi :<nome> <valore>

KEY &key, ogni parametro passato diventa una keyword, da poter usare al momento della chiamata

- vanno sempre tutti dichiarati, dopo quelli obbligatori

10 si effettua con **read** e **print**.

- **read** legge un intero oggetto lisp
- **print** stampa rispettando la sintassi
- **format** formatta l'output
- **t** è lo standard output, uno dei 3 stream standard (con input e error)
- si scrive e si legge su uno stream
- **with-open-file** associa uno stream a un file (e lo chiude alla fine)

AMBIENTE LISP si basa sulla **REPL**: read-eval-print-loop.

APPLY definita come **apply**: funzione $\text{list} \rightarrow \text{exp}$, ovvero prende un designatore di funzione e ritorna un valore

- invece **eval**: $\text{sexp env} \rightarrow \text{sexp}$

CHIUSURA è una struttura generata da una Lambda e che contiene:

- il corpo di lambda
- la lista di parametri formali
- l'ambiente in cui è stata costruita Lambda (**static link** a dove recuperare i valori delle variabili)

AMBIENTI manipolazione di mappe simboli-valori, un ambiente è una sequenza di frames, e un frame è una lista di coppie prefissa dal simbolo frame.

RISCRITTURA Lisp riscrive implicitamente:

- COND in IF
- LET in LAMBDA

INTERPRETE META-CIRCOLARE è l'interprete Lisp, diviso in:

- env operazioni riguardanti la manipolazione di frames e environments
- imc l'interprete "vero e proprio"
- repl un semplice red-eval-print loop