Linguaggi di Programmazione:[l]

Linguaggio di programmazione: notazione non ambigua con cui si descrive un procedimento computazionale.

Insieme di passi che una macchina compie per risolvere un problema, (algoritmo).

Mezzo di comunicazione fra una persona che vuole risolvere un problema ed il computer che si vuole usare per risolverlo.

Programma: (o procedura) realizzazione di un algoritmo in un particolare linguaggio di programmazione.

Definizione precisa e non ambigua del linguaggio (**sintassi**): determina l'insieme dei programmi legali.

Sintassi di un linguaggio: l'insieme di regole che le frasi (i programmi) del linguaggio devono rispettare (come nel linguaggio naturale).

Semantica di un linguaggio: si occupa del significato delle frasi (significato di un programma, delle istruzioni) del linguaggio.

l C. Batini et alii: "Fondamenti di Programmazione dei Calcolatori Elettronici", Franco Angeli, 1990 (cap. l)

Linguaggi di programmazione:

Sintassi:

regole sintattiche (BNF, EBNF, diagrammi sintattici)

Esempi:
<naturale> ::= 0 | <cifra-non-nulla> {<cifra>}
<cifra-non-nulla> ::= 1|2|3|4|5|6|7|9
<cifra> ::= 0 | <cifra-non-nulla>

Semantica:

esprimibile in termini di

- azioni (**semantica operazionale**)
- funzioni matematiche (**semantica denotazionale**)
- formule logiche (**semantica assiomatica**).

Realizzazione:

macchina (fisica o astratta) che sia in grado di eseguire i programmi del linguaggio. Realizzazione di un "traduttore" che renda i programmi eseguibili su un dato elaboratore (**compilatore** o **interprete**).

Un programma per risolvere un problema è più facile da ottenere e più naturale se il linguaggio di programmazione è "vicino" al problema.

Gerarchia di linguaggi di programmazione in base alla “indipendenza” dalla macchina.

- Linguaggi Macchina;
- Linguaggi Assembler;
- Linguaggi di Alto Livello;
- Linguaggi orientati al Problema (query languages ecc...)

Un po' di storia:

1930-40	Macchina di Turing Diagrammi di flusso (Von Neumann)
1940-50	Linguaggi macchina ed ASSEMBLER
1950-55	FORTRAN (Backus, IBM)
1959	COBOL
1960	APL
1950-60	LISP (McCarthy)
1960-65	ALGOL'60 (blocco, stack)
1965	PL/I
1966	SIMULA (tipo di dato astratto, classe)
1970-71	PASCAL (Wirth)
1980	MODULA (Wirth)
1973	PROLOG (Kowalski - Colmerauer)
1975	SETL
1980	SMALLTALK (oggetti)
1983	C++ (C con oggetti)

Il linguaggio macchina:

Descriviamo un semplice linguaggio che consente la programmazione di una macchina di von Neumann.
Il linguaggio (volutamente "giocattolo") appartiene alla categoria dei **linguaggi macchina** cioè linguaggi direttamente eseguibili.

Istruzioni:

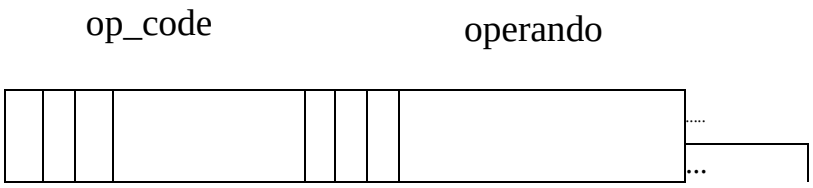
Si dividono in due parti: un **codice operativo** (sempre presente) ed uno o più **operandi** (opzionali).

Il codice operativo specifica l'operazione da compiere, mentre gli operandi le celle di memoria a cui si riferiscono le operazioni. Per semplicità, un solo operando.

Anche le istruzioni sono inserite in memoria e quindi sono memorizzate in binario.

Istruzioni: formato

Un solo operando, per semplicità



s, lunghezza di un'istruzione (in bit)
s=n+m

n, numero di bit dedicati al codice operativo
m, numero di bit dedicati all'indirizzamento degli operandi.

Insieme di istruzioni del linguaggio: al piu' 2^n istruzioni diverse (ciascuna ha un diverso op_code)

Memoria indirizzabile: al più 2^m celle di memoria diverse.

Ulteriore ipotesi semplificativa:
Ciascuna istruzione occupa esattamente una cella di memoria.

RI, registro indirizzi
RD, registro dati
PC, program counter (prossima istruzione)
IR, instruction register (istruzione corrente)
A,B, registri ausiliari

Istruzioni

L'esecuzione di ogni istruzione richiede tre fasi:
1) acquisizione dalla memoria centrale (fetch);
2) interpretazione del codice operativo (decode);
3) esecuzione (execute).

Principali istruzioni:

LOAD = caricamento di una cella di memoria in un opportuno registro ausiliario (consideriamo solo i registri A e B).
STORE = carica il contenuto di un registro in una cella di memoria
READ = lettura da una periferica
WRITE = scrittura su una periferica
Istruzioni numeriche: ADD, DIF, MUL, **DIV**
Gli operandi sono in A e B, il risultato è trasferito nel registro A (DIV, resto in B).
Istruzione di salto incondizionato: JUMP
Modifica l'esecuzione sequenziale del programma
Istruzione di salto condizionato: JUMPZ
Effettua il salto solo se il contenuto di A è zero (utilizza il registro PSW per eseguire il test)
NOP, fa trascorre un ciclo istruzione senza svolgere alcuna operazione (attesa)
HALT, termina 1'esecuzione del programma.

(elenco molto povero)

Poiche' sono 14 istruzioni (VAX della Digital 304!) sono sufficienti 4 bit.

op_code	**istruzione**
0000	LOADA
0001	LOADB
0010	STOREA
0011	STOREB
0100	READ
0101	WRITE
0110	ADD
0111	DIF
1000	MUL
1001	DIV
1010	JUMP
1011	JUMPZ
1100	NOP
1101	HALT

Recentemente sono state proposte macchine RISC (Reduced Instruction Set Computer) caratterizzate dal disporre di un ridotto set di istruzioni con formati regolari eseguite in modo molto efficiente.

Prestazioni migliori rispetto a macchine con molte istruzioni.

Programma

Consiste in due parti: istruzioni e dati. Per semplicità facciamo partire il programma dalla prima cella di memoria (loader). Poi seguono i dati.

Il linguaggio ASSEMBLER

E' difficile leggere e capire un programma scritto in forma binaria.

Linguaggi di programmazione ad alto livello (il piu' possibile indipendenti dalla macchina).

Primo passo: Linguaggi assemblatori.

Le istruzioni corrispondono univocamente a quelle macchina, ma vengono espresse tramite parole chiave.

I riferimenti alle celle di memoria sono fatti mediante nomi simbolici.

Il programma prima di essere eseguito deve essere tradotto in linguaggio macchina (**assemblatore**).

Esempio programma Assembler:

```
      READ    X
      READ    Y
      LOADA   X
      LOADB   Y
      MUL
      STOREA  X
      WRITE   X
      HALT
X     INT
Y     INT
```

Linguaggi di alto livello:

Sono simbolici ed indipendenti dalla macchina dalla macchina (astrazione).

Hanno costrutti "strutturati" (maggiore facilità nello sviluppo e debugging).

Richiedono opportuni "traduttori"

Sono realizzati in termini di istruzioni di basso livello, direttamente eseguite dal processore, attraverso:
- interpretazione (ad es. BASIC)
- compilazione (ad es. FORTRAN; Pascal)

I **compilatori** traducono un programma dal linguaggio L a quello macchina (per un determinato elaboratore).

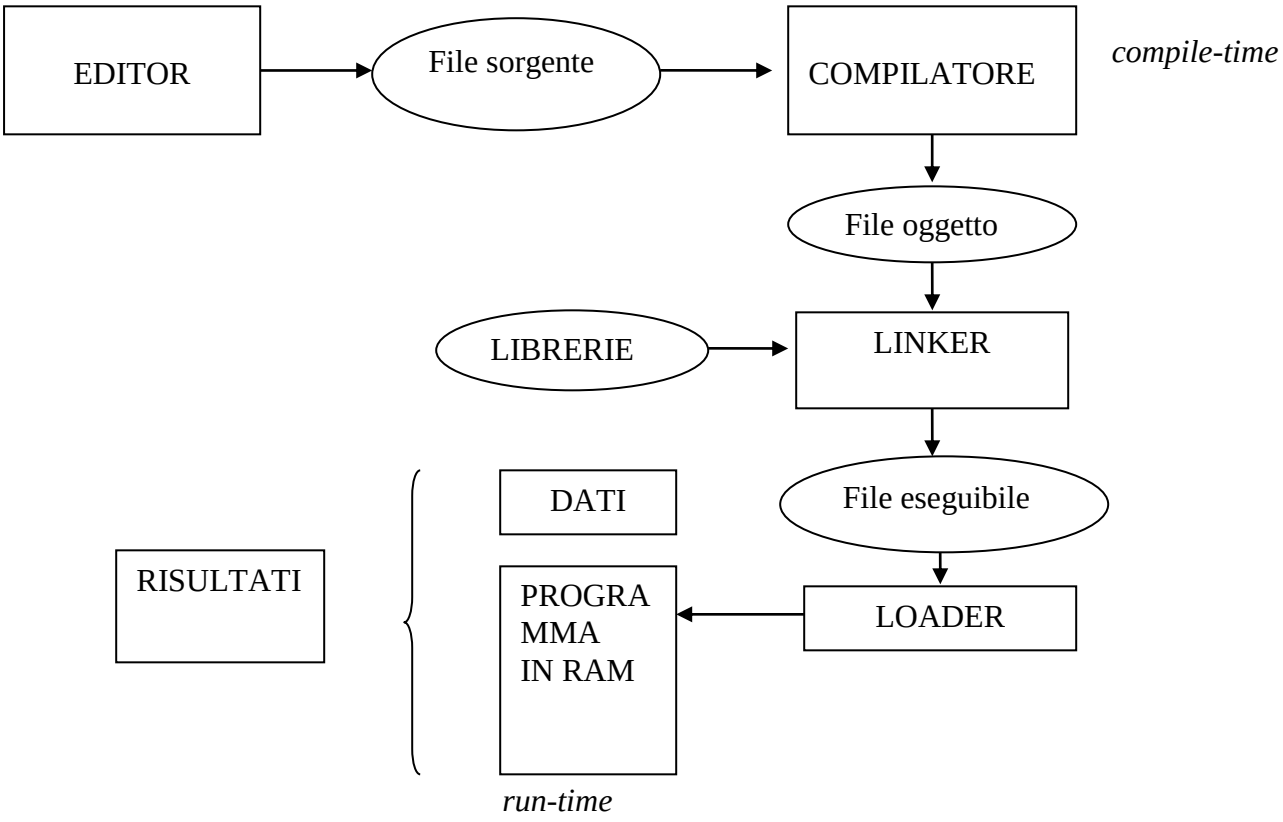
Gli **interpreti** sono programmi capaci di eseguire direttamente un programma in linguaggio L istruzione per istruzione.

I programmi compilati sono in generale piu' efficienti di quelli interpretati.

compile-time: momento in cui avviene la conversione da programma sorgente a programma oggetto.

run-time: momento in cui viene eseguito il programma oggetto.

Approccio compilato:



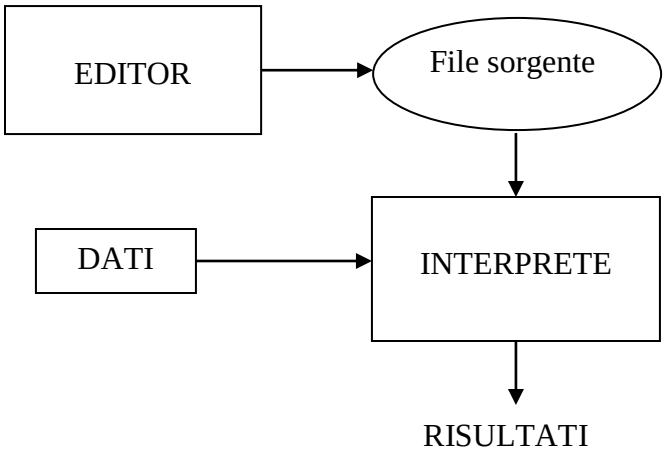
L'ambiente TurboPascal e' un ambiente integrato che nasconde alcuni dei passaggi (ad esempio, linking) .

File sorgente: prog.PAS

File eseguibile: prog.EXE

La fase di caricamento (loading) e' svolta dal sistema operativo.

Approccio interpretato:



L'interprete processa una forma interna del programma sorgente ed i dati nello stesso momento (run-time) e non genera quindi nessun codice oggetto.

Alcuni interpreti analizzano ogni istruzione del programma sorgente ogni volta che viene eseguita. E' ovviamente un approccio molto dispendioso.

Un approccio più efficiente consiste nell'applicare tecniche di compilazione per produrre un codice intermedio che viene poi interpretato dall'interprete.

Semantica di un linguaggio di programmazione:

Attribuisce un significato ai costrutti linguistici del linguaggio.

Molto spesso non è definita formalmente.

Metodi formali:

- ***semantica operazionale***
  azioni

- ***semantica denotazionale***
  funzioni matematiche

- ***semantica assiomatica***
  formule logiche

Benefici per il programmatore (comprensione dei costrutti, prove formali di correttezza), l'implementatore (costruzione del traduttore corretto), progettista di linguaggi (strumenti formali di progetto).

Metodo operazionale:

Definire una semantica operazionale per un linguaggio di programmazione L significa definire una macchina astratta e definire come l'esecuzione delle istruzioni di L viene condotta su tale macchina.

Macchina astratta caratterizzata da uno ***stato***.

Semantica di ciascun costrutto data in termini di ***transizione di stato***.

Il ***significato di un programma*** è rappresentato dalla ***sequenza di stati*** che la macchina astratta attraversa durante l'esecuzione.

(sequenza infinita, il programma non termina)

Vantaggio: ci si basa solo sul programma

Esempio:

Linguaggio con istruzioni di lettura e scrittura, assegnamento di valori a variabili, istruzione composta, istruzione condizionale ed istruzione di iterazione:

readln(x) **writeln** (x)

X:= <espressione>

begin <istruzione> {; <istruzione>} **end**

if <espressione-bool> **then** <istruzione> **else** <istruzione>

while <espressione-bool> **do** <istruzione>

Stato della macchina astratta M:

s = <is, os, mem>

is = sequenza di elementi di input (dati di ingresso)
os = sequenza di elementi di output (risultati)
mem = insieme di coppie <nome, valore>
(configurazione della memoria)

Transizione di stato per M:

s -- > s'

<is, os, mem> --> <is', os', mem'>

(almeno uno tra is', os', mem' differisce dal precedente) .

Operazioni elementari della macchina astratta:

primo(is)	restituisce il primo elemento della sequenza di ingresso
resto(is)	restituisce il resto della sequenza di ingresso tolto il primo elemento (errore se sequenza vuota)
append(v, os)	aggiunge l'elemento v alla sequenza di uscita os
mod(mem, <n,v>)	modifica mem aggiungendovi la coppia <n,v> (se coppia <n,...> gia' presente, la rimpiazza)
val(n,mem)	restituisce l'elemento v se la coppia <n,v> è in mem, altrimenti errore.

Stato corrente, s=<is, os, mem>

Operazione e semantica associata:

S(readln(n)) =
s ---> <resto(is), os, mod(mem, <n,primo(is)>)>

S(writeln(n)) =
s ---> <is, append(val(n,mem), os), mem>

S(n:=exp) =
s ---> <is, os, mod(mem, <n,v>)
se v è il valore risultante dalla valutazione di exp

S(i1; i2) =
s ---> s”
se S(i1) = s ---> s' e
S(i2) = s' --- > s"

S(if bool then il else i2) =
S(i1) se bool=true
S(i2) se bool=false

S(while bool do i) =
s=s0 ---> sl ---> s2 ---> ... ---> sn=s'
per ogni Sj (j<n), bool=true e
per sn, bool=false e
S(i(j)) = sj --->sj+ 1

Metodo denotazionale:

Definire una semantica denotazionale (funzionale) significa fornire un metodo rigoroso per associare, ad ogni programma del linguaggio L la ***funzione calcolata dal programma****:*

Definizione del significato attraverso tipi e procedure viste come funzioni di trasformazione sullo stato del programma

Vantaggio: specifica matematicamente precisa e indipendente dall'architettura

Programma P che termina sull'ingresso i e produce l'output o:

$$fp: fp(i)=o$$

fp viene determinata risolvendo un insieme di ***equazioni funzionali***.

Studiando fp si ricavano proprietà sul programma P (ad esempio, fp totale allora P termina).

Metodo assiomatico: (Hoare)

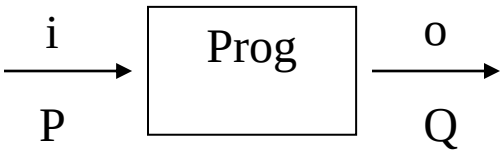
Definire una semantica assiomatica significa fornire un metodo rigoroso per associare ad ogni programma del linguaggio L la ***relazione calcolata dal programma*** (formula logica).
Partendo dunque da una specifica formale di quello che il programma deve fare, prova rigorosa della correttezza attraverso dimostrazione logica
Vantaggio: si possono scoprire proprietà del programma

Programma P che termina sull'ingresso i e produce l'output o:

$$Rp: Rp(i,o)$$

Rp viene generalmente espressa nel calcolo dei predicati.

Lega ***pre-*** **e** ***post-condizione:***



Se la precondizione P e' vera sui dati di ingresso i ed il programma Prog termina, allora la postcondizione Q e' vera sui dati di uscita o.

Semantica assiomatica di Prog, coppia P, Q tale che vale 1'espressione:

$$\{P\}\ Prog\ \{Q\}$$

L'esecuzione di Prog in uno stato che soddisfa P porta ad uno stato che soddisfa Q.

Per ciascuna istruzione del linguaggio occorre definire come sono correlate pre- e post-condizioni. Specificato attraverso ***assiomi*** o ***regole di inferenza.***

Pre- e post-condizione, situazione della memoria prima e dopo l'esecuzione dell'istruzione.

Operazione e semantica associata:

{P} **readln**(n) {P [i/n] }

{P} **writeln**(n) {P}

{P} n:=exp {P[v/n]}

dove v e' il valore risultante dalla valutazione di exp

$$\frac{\{Pl\}\ i1\ \{P2\} \qquad \{P2\}\ i2\ \{P3\}}{\{P1\}\ i1;\ i2\ \{P3\}}$$ Sequenza

$$\frac{\{P1 \text{ AND } v(bool)=true\}\ i1\ \{P2\} \quad \{P1 \text{ AND } v(bool)=false\}\ i2\ \{P2\}}{\{Pl\}\ \text{if bool then i1 else i2}\ \{P2\}}$$ Selezione

$$\frac{\{P \text{ AND } v(bool)=true\}\ i\ \{P\}}{\{P\}\ \text{while bool do i}\ \{P \text{ AND } v(bool)=false\}}$$ Iterazione

Se P è vera prima di eseguire l'istruzione while e questa termina, P è vera anche dopo (ed a questo punto bool ha valore falso).

P, ***invariante del ciclo***.

Il metodo assiomatico viene utilizzato per eseguire ***prove formali*** della correttezza di un programma.

Dimostrare che Prog è corretto rispetto alle condizioni P, Q significa dimostrare che 1'espressione:

$$\{P\}\ Prog\ \{Q\}$$

è soddisfatta usando assiomi e regole di inferenza che definiscono la semantica del linguaggio.

Analisi di programmi: la correttezza

Oltre il 50% del costo di un progetto software e' dovuto alla porova *(test)* ed alla correzione del programma.

Tipologie di errori:

Errori nell'uso del linguaggio:

lessicali,
sintattici
semantici.

lA ---> identificatore

x:=a:=b

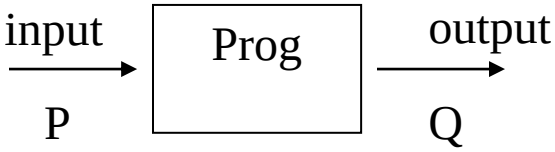
X:=Y con X non dichiarata (semantica statica)

```
var I:0..10;
…
    I:=Y;  {Y potrebbe avere valore >10} (semantica dinamica)
```

Errori logici:
Sono dovuti ad una scelta sbagliata dell'algoritmo o delle strutture dati.

Errori di codifica:
Ad esempio, errore condizione booleana di un ciclo while.

Problema, A



Perchè Prog possa essere utilizzato correttamente i dati di ingresso devono soddisfare la condizione P (vincoli sull'input).

Prog risolve correttamente il problema se l'output soddisfa il vincolo Q.

{P} Prog {Q}, ***specifica***.

La specifica e' soddisfatta se per ogni insieme dei dati di ingresso che soddisfa P, Prog termina ed i dati di uscita soddisfano Q. Si dice che Prog e' **corretto**.

Esempio: r=mcd(x,y) con x, y interi non negativi e non entrambi nulli

P= {x,y interi non negativi} and ((x>0) or (y>0))
Q= {r divide x} **and** {r divide y} **and** {ogni intero
che divide sia x che y divide anche r}

Prove formali di correttezza:

1. Si determinano P e Q (pre-, post-condizione);
2. Si verifica se la specifica {P} Prog {Q} è soddisfatta;
3. Si stabilisce se Prog termina con ogni configurazione dei dati di ingresso che soddisfa P.

Facilitata dall'uso sistematico della programmazione strutturata (un ingresso ed una uscita).

Esempio:
Calcolo del modulo di due numeri interi non negativi: r=x **mod** y (usando solo +, -, *).

P: x>=0 **and** y>0
Q: deve esistere un intero q: x=y*q+r **and** 0<=r<y

```
program modulo (input, output);
var x,y,q: integer;
begin
      readln(x,y);                  {P: x>=0 and y>0}
      q:=0;
      r:=x;                         {x=y*q+r and r>=0}
      while r>=y do
            begin
                  r:=r-y;
                  q:=q+ l
            end.
      writeln(r)                    {Q: x=y*q+r and 0<=r<y}
end.
```

Correttezza: un metodo piu' pragmatico

Si parla di *test* di un programma.

1. Si determina un insieme di dati di ingresso I;
2. Si esegue il programma con i dati I;
3. Si verificano i risultati ottenuti:
   3.1. se l'esecuzione non termina in un tempo ragionevole oppure i risultati non sono quelli attesi, programma non corretto;
   3.2. se non sono stati rilevati errori, ma si vogliono effettuare altre prove si torna al passo 1. Altrimenti il test termina.

Il test di un programma non permette di stabilire la correttezza (a meno di non provare con tutti i possibili dati di ingresso!).

E' tuttavia il metodo piu' usato in pratica per rilevare errori.

Strategie per effettuare il test:

Metodi per il test di un programma si classificano in:

metodi basati sulle specifiche del programma (o a scatola nera)

metodi basati sulla struttura del programma (o a scatola trasparente)

Nel primo caso, esecuzione con diversi dati di ingresso.

Gli infiniti dati di ingresso sono suddivisi in ***classi di equivalenza****:* per ciascuna classe si prova il programma con almeno un elemento di ingresso.

Trade-off: pochi o molti insiemi di equivalenza

Nel secondo, diverse esecuzioni in modo da provare tutte le parti (istruzioni) del programma ***(copertura del programma)***.

Debugging:

L'individuazione di un errore (debugging) e' necessaria per poterlo correggere.

E' spesso difficile localizzare 1'errore (si conosce il sintomo, ma non 1'istruzione che ne e' la causa).

Diagnosi di programmi

Si ipotizza che 1'errore sia in una certa parte del programma e si cerca di avere il massimo di informazione sulle variabili usate in quella parte del programma.

Ipotesi di errore e verifica se dopo la correzione il malfunzionamento è comparso.

Esistono debugger integrati che consentono di eseguire un programma una riga alla volta (tracing), tenendo contemporaneamente sotto controllo i valori delle variabili (finestra di watching).

Altrimenti, ***metodo delle stampe***.