Linguaggi di Programmazione

E` una notazione con cui e` possibile descrivere gli algoritmi.

- **Programma:**

e` la rappresentazione di un algoritmo in un particolare linguaggio di programmazione.

In generale, ogni linguaggio di programmazione dispone di un insieme di "**parole chiave**" (keywords), attraverso le quali e` possibile esprimere il flusso di azioni descritto dall'algoritmo.

Ogni linguaggio e` caratterizzato da una **sintassi** e da una **semantica**:

- **sintassi**: e' l'insieme di regole formali per la composizione di programmi nel linguaggio scelto. Le regole sintattiche dettano le modalità di combinazione tra le parole chiave del linguaggio, per costruire correttamente istruzioni (**frasi**).
- **semantica**: è l'insieme dei significati da attribuire alle frasi (sintatticamente corrette) costruite nel linguaggio scelto.

Linguaggi di programmazione

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)
1972	C (Ritchie)
1973	PROLOG (Kowalski - Colmerauer)
1975	SETL
1980	MODULA, SMALLTALK (oggetti)
1983	C++ (C con oggetti)
1995	JAVA (oggetti)

Il linguaggio macchina

Descriviamo un semplice linguaggio *macchina* per la programmazione di una macchina di Von Neumann.

☞ Il linguaggio macchina è direttamente eseguibile dall'elaboratore, senza nessuna traduzione.

Istruzioni:

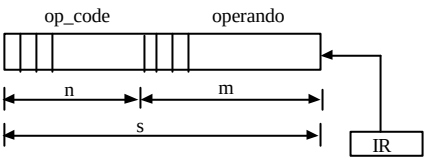
Si dividono in due parti: un **codice operativo** ed, eventualmente, uno o più **operandi**:

- Il **codice operativo** specifica l'operazione da compiere
- gli **operandi** individuano le celle di memoria a cui si riferiscono le operazioni.

Per semplicità, consideriamo istruzioni **ad un solo operando**.

☞ istruzioni ed operandi relativi al programma in esecuzione sono caricati in memoria e quindi sono memorizzati in forma binaria.

Istruzioni: formato



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 più 2^n istruzioni diverse (ciascuna ha un diverso op_code)

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

Ulteriore ipotesi semplificativa:

Supponiamo che ogni istruzione occupi esattamente una cella di memoria.

Linguaggio Macchina: Istruzioni

Ricordiamo i registri interessati:

AR, registro indirizzi
RD, registro dati
PC, program counter (prossima istruzione)
IR, instruction register (istruzione corrente)
A,**B**, registri accumulatori

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

Ogni fase e` realizzata da una sequenza di **microistruzioni**, ciascuna delle quali corrisponde ad un trasferimento di dati tra registri, tra memoria e registri, oppure tra periferiche e registri.

Fetch:

- è realizzata dalla sequenza di microistruzioni:

```
PC -> AR
read(MEMORIA,AR) -> RD
RD -> IR
PC + 1 -> PC
```

- **read** trasferisce dalla memoria centrale (indirizzo AR) al registro RD.

Linguaggio Macchina Principali istruzioni

LOAD caricamento di una cella di memoria in un opportuno registro ausiliario (consideriamo solo i registri A e B: LOADA, LOADB)

Esecuzione di LOADA IND1

```
op(IR) -> AR
read(MEMORIA,AR) -> RD
RD -> A
```

☞ op(IR) seleziona l'operando dell'istruzione contenuta in IR

STORE carica il contenuto di un registro in una cella di memoria (consideriamo solo i registri A e B: STOREA, STOREB)

Esecuzione di STOREA IND1

```
A -> RD
op(IR) -> AR
write(MEMORIA,AR,RD)
```

☞ **write** trasferisce il contenuto del registro RD in memoria centrale (indirizzo AR).

READ trasferimento di dati da una periferica alla memoria centrale

Esecuzione di READ IND1:

supponiamo che RDP sia un registro dati della periferica considerata:

```
RDP -> RD        {Int. periferica -> CPU}
op(IR) -> AR
write(MEMORIA,AR,RD)
```

WRITE scrittura su una periferica

Esecuzione di WRITE IND1

```
op(IR) -> AR
read(MEMORIA,AR)->RD
RD -> RDP
```

Istruzioni aritmetiche: ADD, DIF, MUL, DIV
Gli operandi sono in A e B, il risultato e` trasferito nel registro A (DIV, resto in B).

Istruzioni di salto

Modificano l'esecuzione sequenziale del programma: la prossima istruzione da eseguire non è quella immediatamente successiva nel programma, ma quella individuata dall'indirizzo specificato.

JUMP IND1 Salto incondizionato: la prossima istruzione da eseguire è all'indirizzo IND1.

Esecuzione di JUMP IND1

```
OP(IR) -> PC
```

•**JUMPZ I1** Salto condizionato: effettua il salto ad I1 solo se il contenuto di A è zero (utilizza il registro PSW per eseguire il test)

•**NOP** fa trascorrere un ciclo senza svolgere alcuna operazione (attesa)

•**HALT** termina l'esecuzione del programma.

Set di Istruzioni di un elaboratore:

- è l'insieme delle istruzioni che la macchina è in grado di eseguire direttamente.

- **In questo caso:**

14 istruzioni (VAX della Digital 304!)
=> sono sufficienti 4 bit ($2^4 > 14$).

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):

- da un ridotto set di istruzioni
- con formati regolari

=> Prestazioni migliori rispetto a macchine con molte istruzioni.

Indirizzamento:

- Non sempre il campo operando di una istruzione rappresenta l'indirizzo del dato su cui operare. Esistono varie possibilita`:
  - indirizzamento immediato
  - indirizzamento diretto
  - indirizzamento indiretto
  - indirizzamento con registro indice

Indirizzamento immediato:

Il valore dell'operando e` ***gia` contenuto*** nel campo operando. Non e` richiesto alcun accesso alla memoria durante l'esecuzione dell'istruzione.

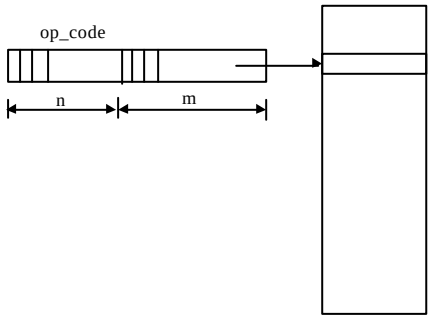
☞ Con istruzioni a formato fisso, gli operandi possono occupare al massimo m bit.

Indirizzamento diretto:

Il campo operando contiene l'indirizzo assoluto di una cella di memoria in cui e` mantenuto il valore di un dato (e` la modalita` vista fino ad ora).

- Per il recupero dell'operando e` richiesto ***un accesso alla memoria***.

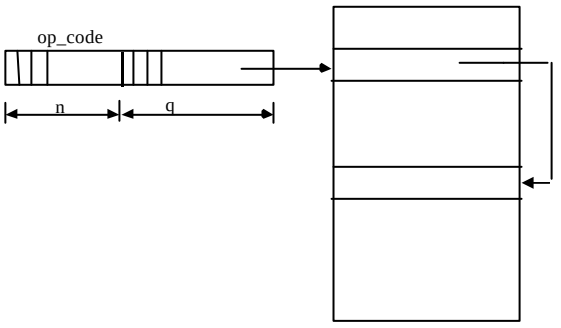
Puo` essere oneroso se la memoria e` molto grande o se e` necessario *rilocare* il programma ed i dati.



- **Indirizzamento indiretto:**

L'***indirizzo dell'operando*** e` contenuto nella parola di memoria indirizzata dal campo operando.

☞ E` richiesto ***un doppio accesso alla memoria*** per recuperare l'operando.
☞ Si possono usare piu` bit per rappresentare l'indirizzo dell'operando.

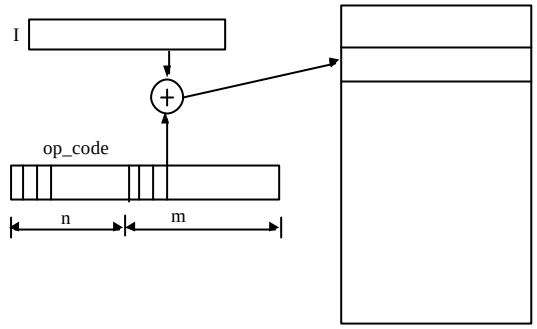


Indirizzamento mediante registro indice:

L'indirizzo dell'operando si ottiene **sommando** algebricamente il contenuto del campo operando dell'istruzione e quello di un particolare registro della CPU (detto **registro indice**).

- ☞ Richiede **un unico accesso** alla memoria per recuperare l'operando ed ***una operazione di somma***.

E` particolarmente efficiente se si deve accedere ad un certo numero di operandi memorizzati in celle di memoria consecutive (ad esempio, vettori in C).



Gestione piu` semplice in caso di rilocazione.

Il linguaggio ASSEMBLER

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

0 0100	0000	0000	1000
1 0100	0000	0000	1001
2 0000	0000	0000	1000
3 0001	0000	0000	1001
4 1000	0000	0000	0000
5 0010	0000	0000	1000
6 0101	0000	0000	1000
7 1101	0000	0000	0000
8 0000	0000	0000	0000
9 0000	0000	0000	0000

- **Linguaggi assemblatori (Assembler):**
  - • Le istruzioni corrispondono univocamente a quelle macchina, ma vengono espresse tramite nomi simbolici (parole chiave).
  - • I riferimenti alle celle di memoria sono fatti mediante nomi simbolici (identificatori).
  - • Identificatori che rappresentano **dati** (costanti o variabili) oppure istruzioni (**etichette**).
  - ☞ 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 programmazione di alto livello

- **Linguaggio Macchina:**
  - conoscenza dei metodi di rappresentazione delle informazioni utilizzati.
- **Linguaggio Macchina ed Assembler:**
  - necessita` di conoscere dettagliatamente le caratteristiche della macchina (registri, dimensioni dati, set di istruzioni)
  - semplici algoritmi implicano la specifica di molte istruzioni
- **Linguaggi di Alto Livello:**

Il programmatore puo` astrarre dai dettagli legati all'architettura ed esprimere i propri algoritmi in modo simbolico.

☞ Sono indipendenti dalla macchina (astrazione).

- **Esecuzione:**

Sono tradotti in sequenze di istruzioni di basso livello, direttamente eseguite dal processore, attraverso:

- interpretazione (ad es. BASIC)
- compilazione (ad es. C, FORTRAN, Pascal)

Sintassi di un linguaggio di programmazione

Dato un linguaggio:

- V, **alfabeto**, vocabolario o lessico. E` l'insieme dei simboli con cui si costruiscono i programmi (ad esempio, parole chiave o singoli caratteri).

- V*, **universo linguistico** su V. E` l'insieme di tutte le sequenze finite di lunghezza arbitraria di elementi di V.

Gli elementi di V* sono le **frasi** o stringhe di V.

- **Esempio:**

V={if, else, ==, A, 0, =, +, 1, 2, (,)}

V*={**if (A == 0) = A :== A + 1 else A = A + 2**,
if (A == 0) A = A + 2,
A = A + A,
if else A,
do =A,
...}

Un **linguaggio L** sull'alfabeto di V e' un sottoinsieme di V*.

Grammatiche

La sintassi di un linguaggio puo` essere descritta in modo informale (ad esempio, a parole) oppure in modo formale.

Grammatica formale:

e` una notazione matematica che consente di esprimere in modo rigoroso la sintassi dei linguaggi di programmazione.

Def: Grammatica BNF (Backus-Naur Form)

Una grammatica BNF e` un insieme di 4 elementi:

- un alfabeto terminale V
- un alfabeto non terminale N
- un insieme finito di regole (produzioni) P del tipo:
  X::=A
  dove X ∈ N, ed A e` una sequenza di simboli α (*stringhe*): α∈ (N ∪ V).
- un assioma (o simbolo iniziale) S

- **Esempio:**

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

BNF

Una BNF definisce un linguaggio sull'alfabeto terminale, mediante un meccanismo di derivazione o riscrittura:

- **Def:**

Data una grammatica G e due stringhe β, γ elementi di $(N \cup V)^*$, si dice che γ **deriva direttamente da** β ($\beta \rightarrow \gamma$), se le stringhe si possono decomporre in:

$\beta = \eta A \delta \qquad \gamma = \eta \alpha \delta$

ed esiste la produzione $A ::= \alpha$.

Si dice che γ **deriva da** β se:
$\beta=\beta 0 \rightarrow \beta 1 \rightarrow \beta 2 \rightarrow ... \rightarrow \beta n = \gamma$

- **Def:**

Data una grammatica G, si dice **linguaggio generato da G**, L_G, l'insieme delle **frasi di V** derivabili, applicando le produzioni, a partire dal simbolo iniziale S.

Le frasi di un linguaggio di programmazione vengono dette ***programmi*** di tale linguaggio.

In una grammatica BNF possono esistere piu` regole con la stessa parte sinistra:

$X ::= A_1$
$X ::= A_2$
....
$X ::= A_N$

Equivale a scrivere:

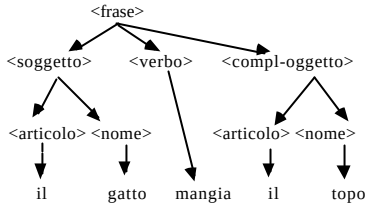
$X ::= A_1 | A_2 | .. | A_N$ (sono tutte alternative).

BNF: esempi

V={il, gatto, topo, sasso, mangia, beve}
N = { <frase>, <soggetto>, <verbo>,
<compl-oggetto>, <articolo>, <nome>}
S = <frase>
P (produzioni):
<frase>::=<soggetto><verbo><compl-oggetto>
<soggetto>::=<articolo><nome>
<articolo>::=il
<nome>::=gatto|topo|sasso|
<verbo>::=mangia | beve
<compl-oggetto>::=<articolo><nome>

Albero sintattico: esprime il processo di derivazione di una frase mediante una grammatica

"il gatto mangia il topo"



Esempio di derivazione (left-most):

A partire dallo scopo della grammatica, si riscrive sempre il non-terminale piu` a sinistra.

<frase>

--> <soggetto><verbo><compl-oggetto>

--> <articolo><nome><verbo><compl-oggetto>

--> ***il*** <nome><verbo><compl-oggetto>

--> ***il gatto*** <verbo><compl-oggetto>

--> ***il gatto mangia*** <compl-oggetto>

--> ***il gatto mangia*** <articolo><nome>

--> ***il gatto mangia il*** <nome>

--> ***il gatto mangia il topo***

Extended BNF: alcune estensioni

X::=[a]B (a puo' comparire zero od una volta).

equivale a:

X::=B|aB

$X:=\{a\}^{n}B$ a puo' comparire da 0 ad un massimo di n volte.

n=3, equivale alla produzione:

X::=B|aB|aaB|aaaB

Se n e` omesso il massimo e` un valore finito arbitrario.

Equivale a:

X::=B|aX (**ricorsiva**)

Per raggruppare categorie sintattiche:

X::= (a | b) D | c

equivale a:

X::= a D | b D | c

EBNF: esempio

V ={0,1,2,3,4,5,6,7,8,9}

N={<naturale>,<cifra>, <cifra-non-nulla>}

S = <naturale>

P e` costituito dalle produzioni:

<naturale>::=<cifra>|<cifra-non-nulla> {<cifra>}
<cifra> ::= 0 | <cifra-non-nulla>
<cifra-non-nulla>::=1|2|3|4|5|6|7|8|9

EBNF : esempi

Numeri interi di lunghezza qualsiasi con o senza segno (non si permettono numeri con piu` di una cifra se quella piu` a sinistra e` 0 es: 059)

V ={0,1,2,3,4,5,6,7,8,9}U{+,-}

N={<intero>, <intero-senza-segno>, <cifra>,
<cifra-non-nulla>}

S = <intero>

P=

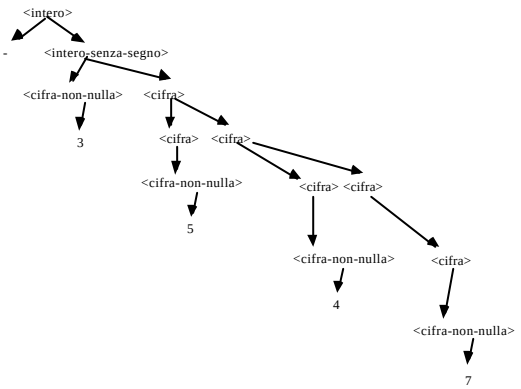
<intero>::=[+|-] <intero-senza-segno>
<intero-senza-segno>::=0|
<cifra-non-nulla> {<cifra>}
<cifra>::= <cifra-non-nulla>|0
<cifra-non-nulla>::=1|2|3|4|5|6|7|8|9

Identificatore:

<identificatore> ::= <lettera> { <lettera> | <cifra>}
<lettera> ::= A | B | ... | Z
<cifra> ::= 0|1|2|3|4|5|6|7|8|9

- **Esempio:**

Albero sintattico per la generazione del numero -3547 usando la grammatica EBNF:



Semantica di un linguaggio di programmazione

Attribuisce un significato ai costrutti linguistici del linguaggio.

Molto spesso non e` 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).