UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

# Appunti di
# Linguaggi di Programmazione I

## Anno 2021

Valentino Bocchetti

# Contents

# 1 Introduzione al corso e informazioni

`Email` → pab@unina.it

`Ricevimento` → da richiedere in precedenza con una mail

## 1.1 Esame

- Uno scritto java + altre parti del corso;
- In caso di dubbi c'è un orale;

Lo scritto può essere mantenuto per la durata di un anno.

# 2 Suddivisione del corso

## 2.1 Parte I

- Storia e concetti di base;
- Primi cenni ai paradigmi dei linguaggi di programmazione;
- Il modello imperativo;

## 2.2 Parte II (modello object-oriented, Java)

- Studio dei costrutti più avanzati
  - qualificatori di classi;
  - metodi e attributi;
  - classi astratte;
  - interfacce;
  - template;
  - classi interne;
  - gestione degli errori;
  - eccezioni;

## 2.3 Parte III (linguaggi funzionali, ML)

- ML e costrutti avanzati;
- funzioni di ordine superiore;
- polimorfismo;
- tipi di dato astratti e interfacce;
- template;
- gestione degli errori;
- eccezioni;
- type inference;

## 2.4 Parte IV (linguaggi logici, Prolog)

- Costrutti fondamentali
  - termini;
  - predicati;
  - regole;
- Tecniche di programmazione avanzate
  - invertibilità;
  - nondeterminismo;

# 3 Lezione del 09-03

## 3.1 Breve storia

Il primo linguaggio introdotto è il FORTRAN (introdotto con lo scopo di calcoli scientifici). Nasce la concezione di variabile

| Anno | Linguaggio |
|---:|---|
| 1954 | FORTRAN (FORmula TRANslation) |
| 1960 | COBOL |
| | ALGOL (60 e 68) |
| | `PL/1` |
| | Simula 67 |
| | PASCAL |
| | LISP |
| | APL |
| | BASIC |
| `1970/1980` | PROLOG |
| | SMALLTALK |
| | C |
| | `MODULA/2` |
| | ADA |

- **Cobol** → orientato ai database (introduce i record);
- **Agol60** → indipendenza dalla macchina e definizione mediante grammatica (supporto dell'iterazione e ricorsione, introduzione delle strutture a blocco);
- **Lisp** → primo vero linguaggio di manipolazione simbolica, linguaggio funzionale (non esiste lo statement di assegnazione);
- **Prolog** → primo (e principale) linguaggio di programmazione logica, porta l'invertibilità e una programmazione in stile nondeterministico (essenzialmente non tipato; queste vengono ottenute mediante metaprogrammazione);
- **Simula 67** → classe come incapsulamento di dati e procedure, istanze delle classi (oggetti);
- `PL/1` abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; (multitasking);
- **Pascal** → programmazione strutturata, tipi di dato definiti da utente (ricchezza di strutture dati); nessuna encapsulation.

## 3.2 Definizione di base

### Linguaggio di programmazione

È un linguaggio che è usato per esprimere (mediante un programma) un processo con il quale un processore può risolvere un problema.

### Processore

È la macchina che eseguirà il processo descritto dal programma (da non intendere come un singolo oggetto, ma coome una *architettura di elaborazione*).

### Programma

È l'espressione codificata di un processo.

## 3.3 Linguaggi completi

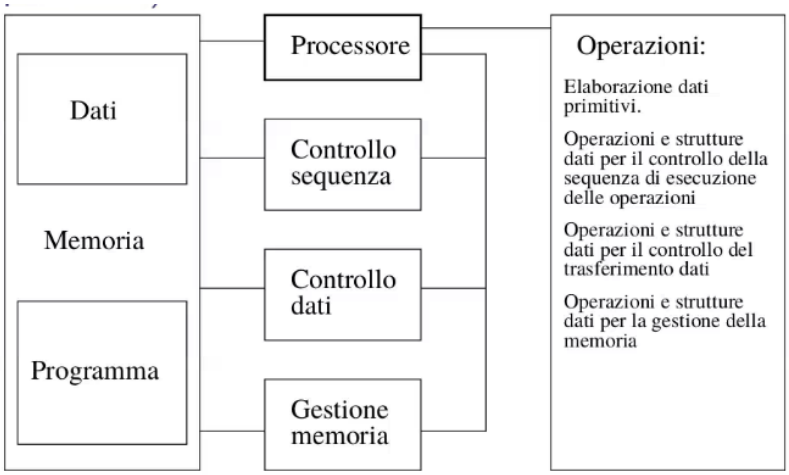
Sono linguaggi in grado di programmare qualunque funzione calcolabile (*general purpose*). Tecnicamente devono poter simulare qualunque *macchina di Turing.*

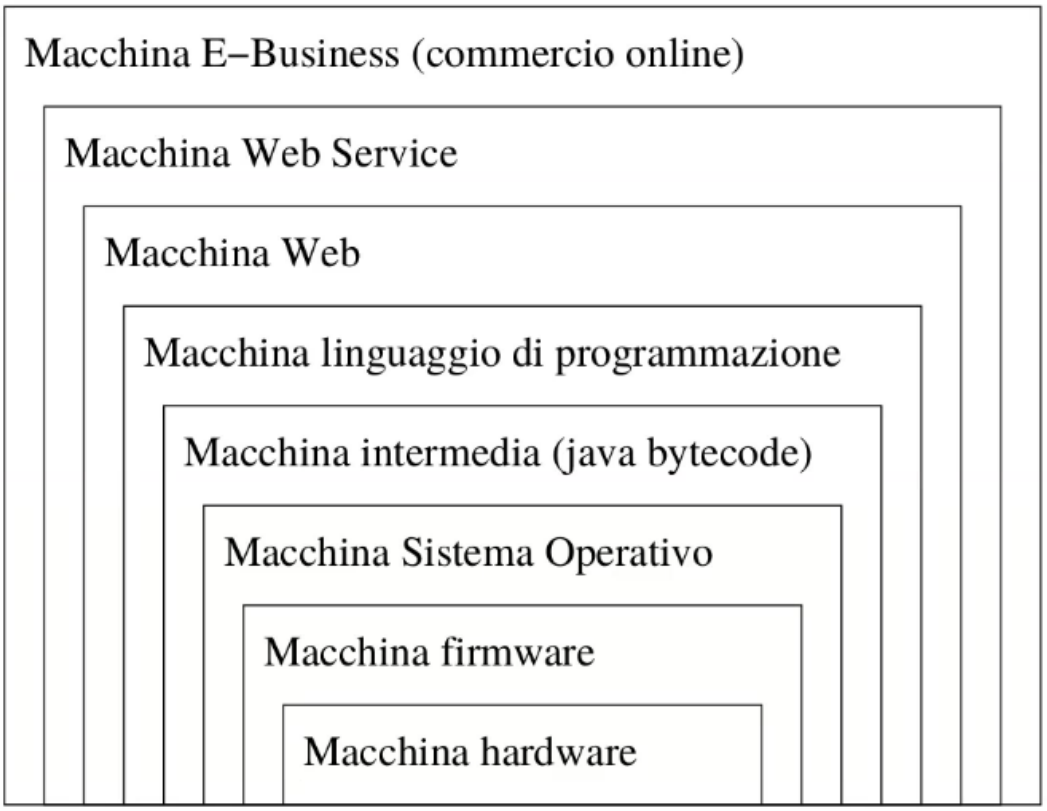
Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione. Per dimostrare la completezza di un linguaggio lo si usa per simulare arbitrarie macchine di Turing.

Ad esempio non sono linguaggi completi **SQL** (perchè la terminazione dei programmi è sempre decidibile; è spesso immerso in linguaggio completo) e **HTML**.

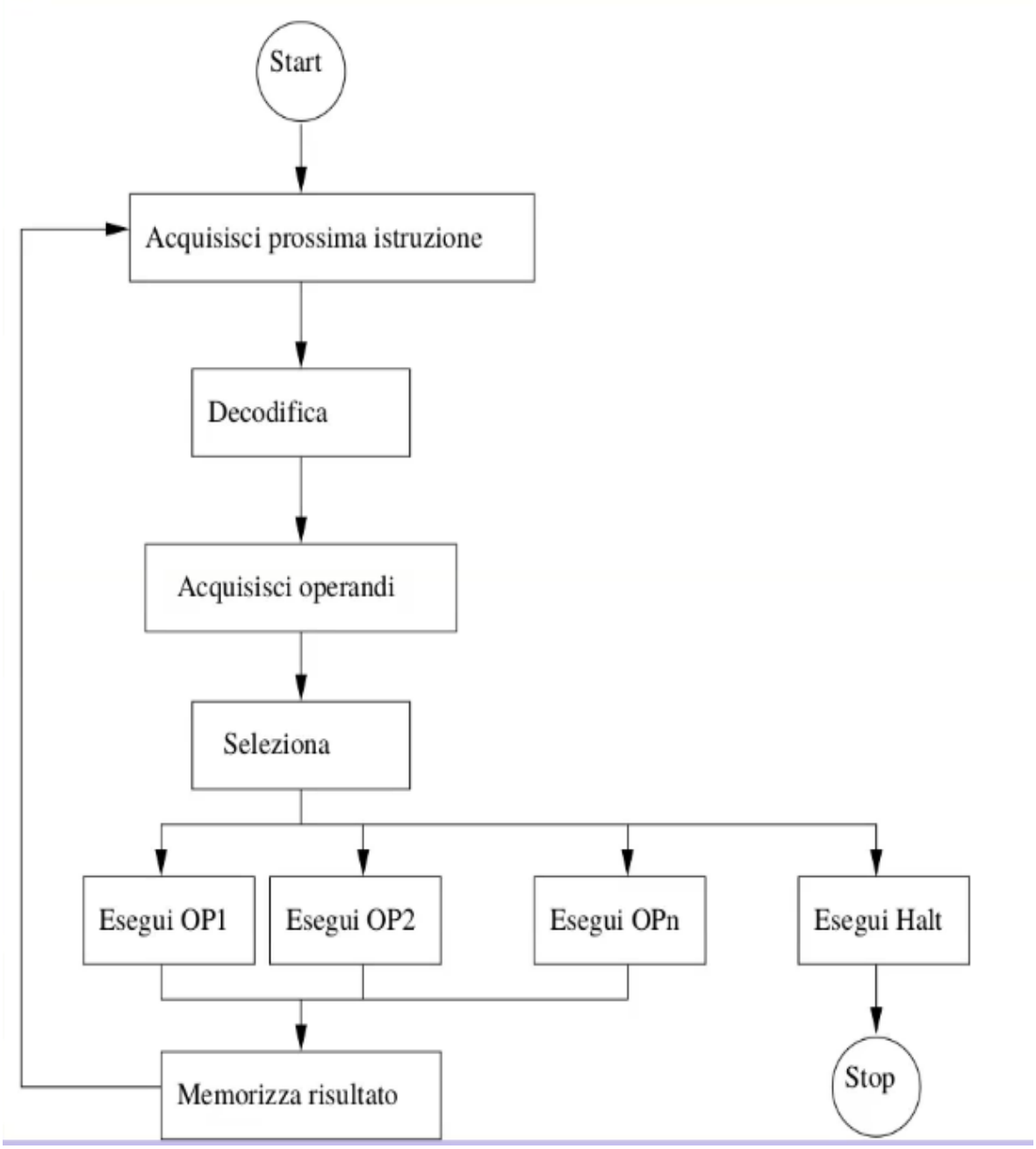
### 3.4 Macchine astratte

Dato un linguaggio di programmazione $L$, una macchina astratta per $L$ (in simboli `M_L`) è un qualsiasi insieme di strutture dati e algoritimi che permettano di memorizzare ed eseguire programmi scritti in $L$. La struttura di una macchina astratta è (essenzialmente memoria e processore):



**Processore della macchina astratta**



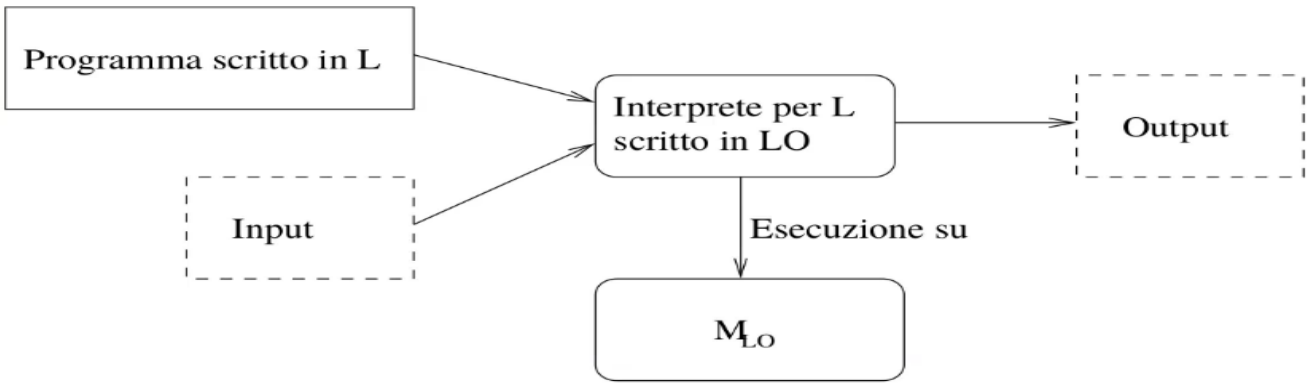

Nel tempo si è pensato a vari modi di implementarlo:

- Hardware (es per *Lisp* e *Prolog*), ma abbandonato rapidamente;
- Firmware → soluzione più flessibile e semplice (ed economico);
- Software → come nel caso della JVM;

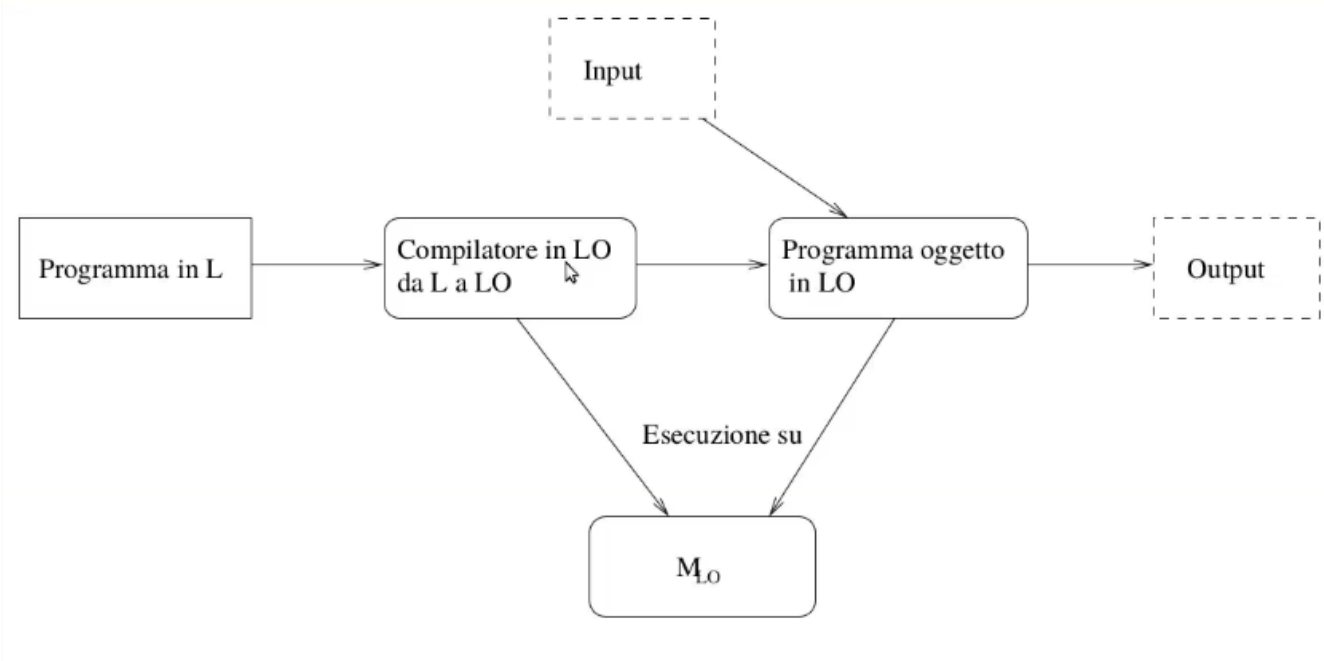
## 3.5 Traduttori del linguaggio

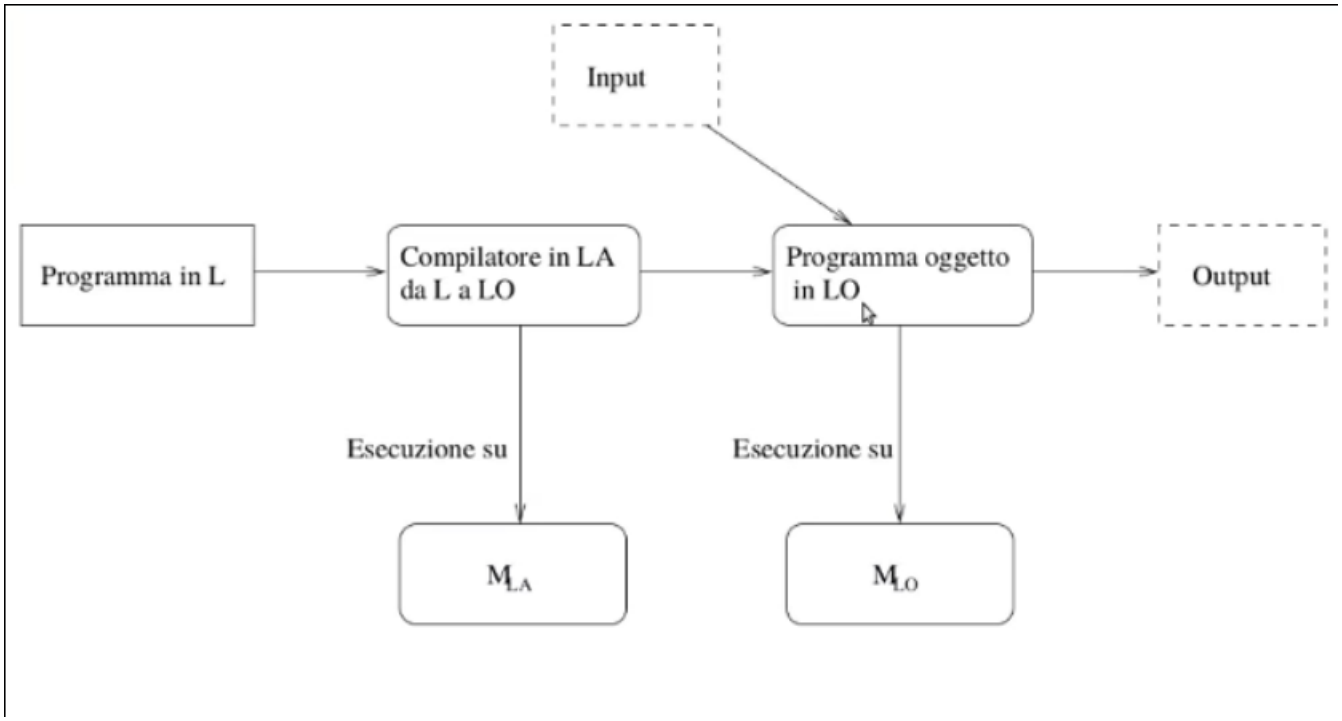
- **Interpreti** → traducono ed eseguono un costrutto alla volta
  - PRO : debug;
- **Compilatori** → prima traducono l'intero programma, che poi può essere eseguito
  - PRO : velocità di esecuzione finale;
  - PRO : più controlli e in anticipo.

**Interpretazione pura**



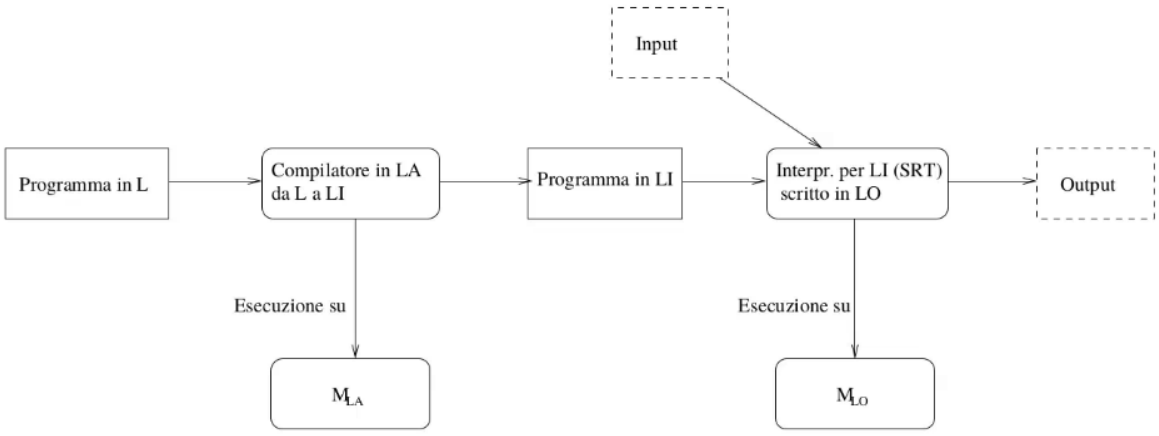

**Compilazione pura** (caso semplice e generale)






Nella compilazione pura può accadere che $L$ ed $LO$ siano uguali (il vantaggio è il riutilizzo del compilatore $\rightarrow$ **custom compilation**)

**Compilazione per macchina intermedia**




($SRT \rightarrow$ supporto al run time)

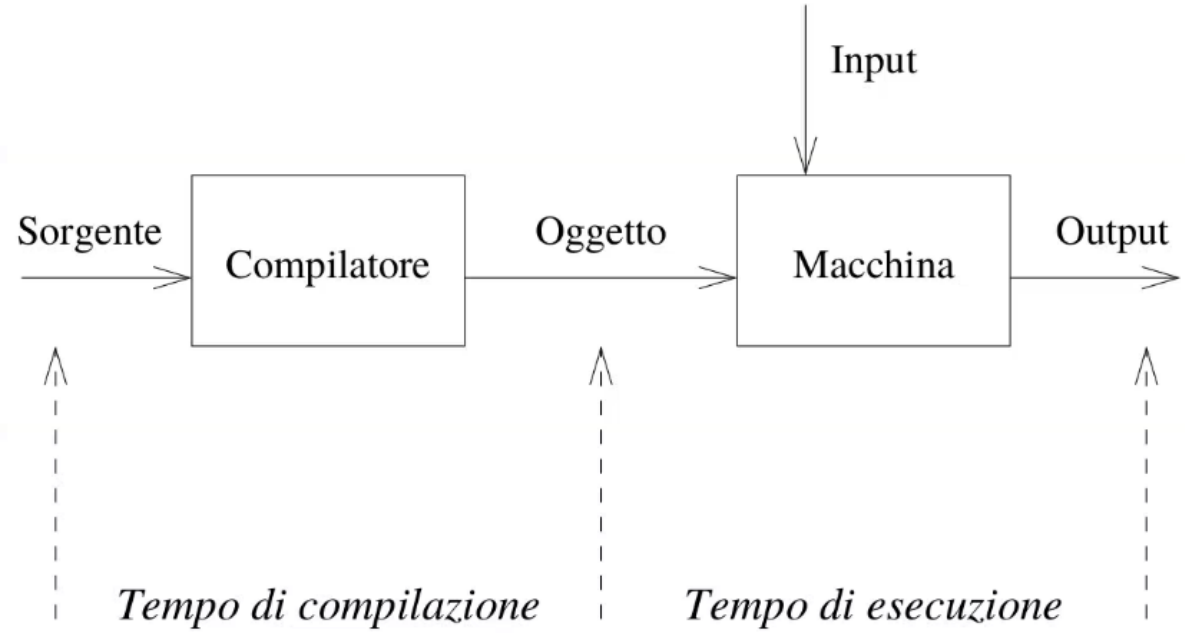
Torna utile nel caso di linguaggi molto diversi dalla macchina astratta sottostante. Invece di creare hardware apposito creiamo un interprete che lo sostituisca

### 3.6 SRT

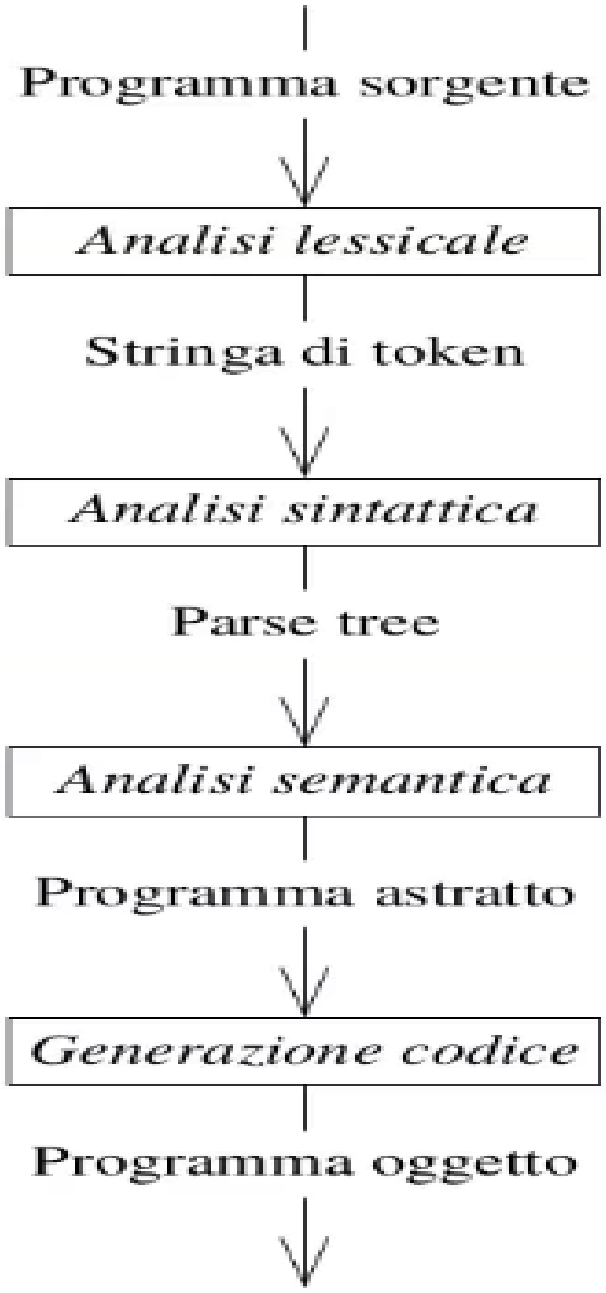
- Funzionalità aggiuntive (rispetto alla macchina astratta sottostante);
- Funzioni di basso livello / interfacce con il S.O. (es `I/O`);
- Gestione della memoria;
- Garbage collection;
- Gestione dell'heap;
- Gestione dello stack;

Non è necessariamente una macchina astratta radicalmente diversa, ma a volte solo un pacchetto di funzioni o librerie aggiunte automaticamente al codice oggetto.

### 3.7 Compilazione ed esecuzione



**Fasi della compilazione**




(Analisi lessicale e sintattica sono stati un problema → venivano fatte in maniera artigianale)

### 3.8 Proprietà dei linguaggi

- **Semplicità** → (coincisione) VS (leggibilità)
  - **Semantica** → minimo numero di concetti e strutture;
  - **Sintattica** → unica rappresentabilità di ogni concetto;
- **Astrazione** → rappresentare solo attributi essenziali
  - **Dati** → nascondere i dettagli di oggetti;
  - **Procedure** → facilitare la modularità del progetto;
- **Espressività** → (facilità di rappresentazione di oggetti) VS (semplicità);
- **Ortogonalità** → meno eccezioni alla regole del linguaggio;
- **Portabilità.**

### 3.9 Criteri di scelta dei linguaggi

- Disponibilità dei traduttori;
- Maggiore conoscenza da parte del programmatore;
- Esistenza di standard di portabilità (es ADA);
- Comodità dell'ambiente di programmazione;
- Sintassi aderente al problema;
- Semantica aderente alla architettura fisica non importante (non è più un criterio stringente).

## 4 Lezione del 12-03

### 4.1 Paradigmi computazionali

Il paradigma di appartenenza può influenzare radicalmente il modo in cui si risolve il problema, ma non è l'unica discriminante. Infatti sono discriminanti anche:

- Il sistema di tipi supportato;
- Eventuale supporto alle eccezioni;
- Modello di concorrenza e sicronizzazione.

Sono presenti tre tipi di paradigmi computazionali:

- Imperativo;
- Funzionale;
- Logico.

*es.* Un programma specifica sequenze di modifiche da apportare allo stato della macchina (memoria)

Vogliamo scrivere in tutti e tre i linguaggi la funzione membro (X, L), che decida se l'elemento X appartiene alla lista L

`membro(2, [1,2,3]) = true; membro(2, [1,2,3]) = false;`

A questo scopo supponiamo siano già esistenti le funzioni:

- vuota(L), che restituisce true se L è vuota, false altrimenti;
- testa(L), che restituisce il primo elemento della lista L;
- coda(L), che restituisce una sottolista ottenuta rimuovendo il primo elemento di L

### 4.1.1 Paradigma imperativo + esempio (pseudo-linguaggio + c):

```
procedure membro (X,L)
    local L1 = L
    while not vuota(L1) and not X=testa(L1)
do L1 = coda(L1)
return not vuota(L1)


bool member(X, L){
    List L1 = L;
    while ( ! empty(L1) && ! X=testa(L1))
L1 = coda(L1);
    return (! vuota(L1));
}
```

### 4.1.2 Paradigma funzionale + esempio (pseudo-linguaggio + lisp):

Il programma e le sue componenti sono funzioni. Esecuzione come valutazioni di funzioni

```
function member(X, L)
if vuota (L) then false
else if X == testa(L) then true
else member(X, coda (L))
```

```
(defun membro (x l)
    (cond ((null l) nil)
((equal x (first)) T)
(T (membro x (rest l)))))
```

Notiamo che anche il c può prestarsi alla versione funzionale

```
bool member(X, L){
    return (vuota (L)) false :
(X == testa(L))? true :
member(X, coda(L))}
```

### 4.1.3 Paradigma logico + esempio (prolog):

Programmma come descrizione logica di un problema. Esecuzione analoga a processi di dimostrazione di teoremi

```
membro(X, [X|L]) .
membro(X, [Y|L]) : - membro(X, L).
```

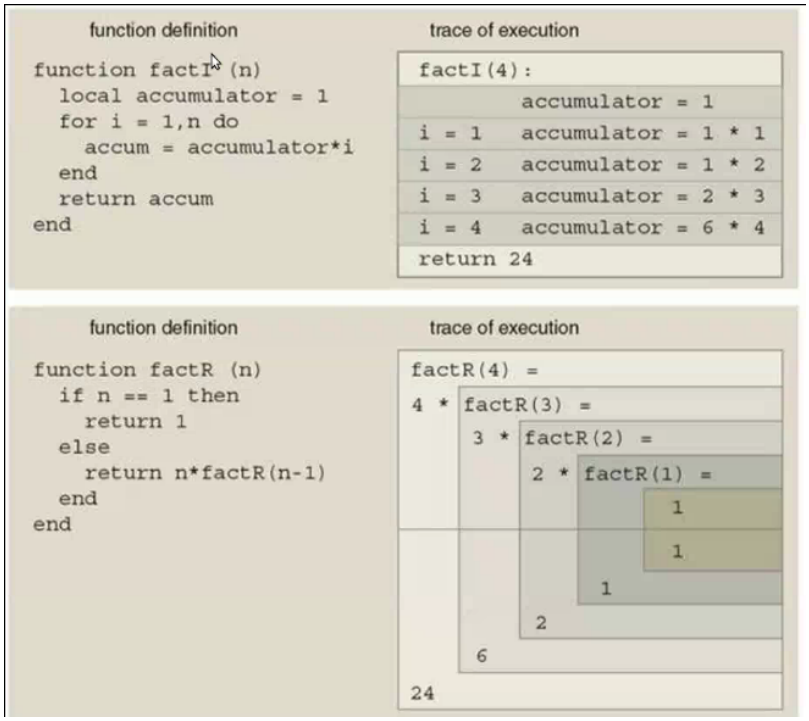
Notiamo come i parametri formali di member possono essere pattern (consistono di definizioni di predicati). Esecuzione:

`member(2,[1,2,3]) restituisce yes (true)` `member(0,[1,2,3]) restituisce no (false)`

Presenta diverse proprietà:

- Si comporta come un generatore → query con variabili: member(X,[1,2,3]) restituisce
  - X=1;
  - X=2;
  - X=3;
  - no;
- Propone risposte con variabili: member(1, L) restituisce
  - `L=[1| L_0], L=[Y_0, 1 | L_1]; L = [Y_0, Y_1, 1 | L_2] ...`
- Invertibilità → nessuna distinzione tra input e output → un solo predicato (programma), molte funzioni;

### 4.1.4 Imperativo vs funzionale




Si noti l'eleminazione degli assegnamenti:

- $n$ rimpiazza $i$ (ricorsione invece di cicli)

### 4.1.5 Orientato ad oggetti

Programma costituito da ogetti che scambiano messaggi.

### 4.1.6 Parallelo

Programmi che descrivono entità distribuite che sono eseguite contemporaneamente e in modo asincrono.

## 4.2 Modello Imperativo nel dettaglio

Consiste in un insieme di "contenitori di dati":

- Ad es. parole (o celle) della memoria centrale;
- Tipicamente rappresentate dal loro indirizzo

Concettualmente la memoria quindi è:

- Una funzione da uno spazio di locazioni ad uno spazio di valori;
- mem(loc) = "valore contenuto in loc"

La sintassi è del tipo

`<assegnazione>` ::= `<name>` `<assigment-operator>` `<expression>`

dove:

- `<name>` rappresenta la locazione dove viene posto il risultato;
- `<expression>` vengono specificati una computazione e i riferimenti ai valori necessari alla computazione

In Pascal:

```
a := b + c;
```

Il valore di `<expression>` va memorizzato nell'indirizzo rappresentato da `<name>`. Il valore di `<expression>` dipenderà dai valori contenuti negli indirizzi degli argomenti di `<expression>` rappresentati dai nomi di questi ottenuto seguendo le prescrizione del codice associato al suo nome

Questo tipo di modello è il più vicino agli elaboratori "standard". I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore (come nel caso del modo di pilotare i display, salvare su memorie periferiche). Ogni esecuzione consiste di 4 passi (in sostanza un assrgnamento):

- Ottenere indirizzi delle locazioni di operandi e risultato;
- Lettura degli operandi;
- Valutazione del risultato;
- Memorizzazione del risultato

In questo tipo di modello le variabili di un programma costituiscono una memoria, mentre i parametri formali delle `funzioni/procedure` no (sia concettualmente che per come sono realizzati). L'ambiente (environment) comprende:

- Un insieme di nomi di variabili e parametri
  - non indirizi di memoria, piuttosto identificatori, associati al loro valore (direttamente e non)

Nel paradigma imperativo, la funzione env (ambiente) associa gli identificatori a locazioni di memoria, le quali, a loro volta, sono associate (funzione mem) al contenuto di memoria. Il valore di una variabile $x$ è mem(env(x)). In questo tipo di paradigma la funzione env identifica una associazione immutabile (la locazione di memoria associata a un nome non cambia)

Nel paradigma funzionale, non esiste la funzione mem e la funzione env associa direttamente gli identificatori al valore della memoria

**ATTENZIONE**: env(x) è immutabile finchè esiste x (gli identificatori nascono e muoiono durante l'esecuzione)

In una assegnazione del tipo `x := x +1;` vediamo che il valore di destra va salvato nella variabile di sinistra:

- la $x$ di sinistra indica la locazione associata al nome (cioè env(x));
- la $x$ di destra indica il valore della memoria (cioè mem(env(x)))

## 4.3 Esempi di semantica di assegnamenti in C

### 4.3.1 Assegnazione

`x=y`

In questo caso avremo:

- sx:(env(x));
- dx: mem(env(y));

### 4.3.2 Operatore &

```
int *x; y
x = &y;
```

In questo caso avremo:

- sx: env(x);
- dx: env(y);

### 4.3.3 Operatore & a sinistra

`&y = x;`

`NB: ERRORE!` → sarebbe come scrivere env(env(y)) che è malformato (env accetta solo cose che siano identificatori)
Il C riporta errore di sintassi *not an lvalue*. Valori corretti possono essere:

- nome di variabile;
- `*(<pointer-expression>)`;

### 4.3.4 Puntatori (a destra)

`y = *x;`

In questo caso avremo:

- sx: env(x);
- dx: mem(mem(env(y)));

### 4.3.5 Puntatori (a sinistra)

In C `*x` indica la cella di memoria puntata da x (la cella di memoria il cui indirizzo sta scritto dentro x)

```
*x = y;
```

In questo caso avremo:

- sx: mem(env(x));
- dx: mem(env(y))

### 4.3.6 vettori e puntatori

```
int v[3];
```

In questo caso env(v) indica la prima cella di memoria del vettore v

y = v[1];

In questo caso avremo:

- sx: env(y);
- dx: mem(env(v) +1);

v[1] = y;

In questo caso avremo:

- sx: env(v) +1;
- dx: mem(env(y));

y = x[1] + v[1]; //puntatore + vettore

In questo caso avremo:

- sx: env(v) +1;

- dx: mem(men(env(x) +1)) + mem(env(v) + 1);

```
x[1] = y
```

In questo caso avremo:

- sx: mem(env(x)) +1

```
y = *x + *v
```

In questo caso avremo:

- dx: mem(mem(env(x))) + mem(env(v))

```
*v = y;
```

In questo caso avremo:

- sx: env(v); (visto come v[0] sarebbe env(v) + 0)

```
y = *(v+2)
```

In questo caso avremo:

- dx: mem(env(v) + 2);

```
y = (*v) + 2
```

In questo caso avremo:

- dx: mem(env(v)) + 2;

```
*(x+2) = ...
```

In questo caso avremo:

- sx: mem(env(x))+2

# 5 Lezione del 16-03

## 5.1 Data Object e legami

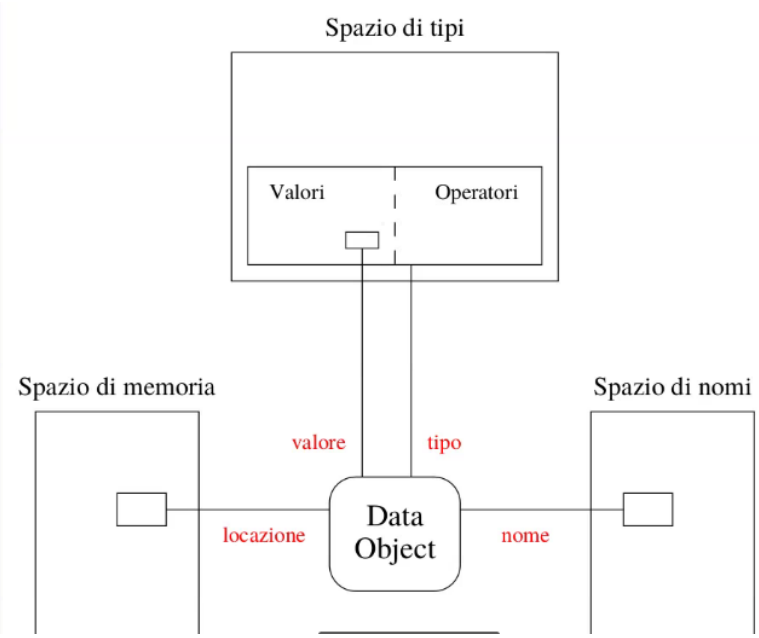
Ogni variabile, parametro, oggetto può essere rappresentato da un Data Object.

È una quadrupla (L, N, V, T) → locazione, nome, valore, tipo.

Legame → determinazioni di una delle componenti

Nel paradigma imperativo:

- Il legame di locazione corrisponde alla funzione *env*;
- il legame di valore di una variabile di nome x corrisponde a *mem(env(x))*.




### 5.1.1 Variazione dei legami

Possono avvenire:

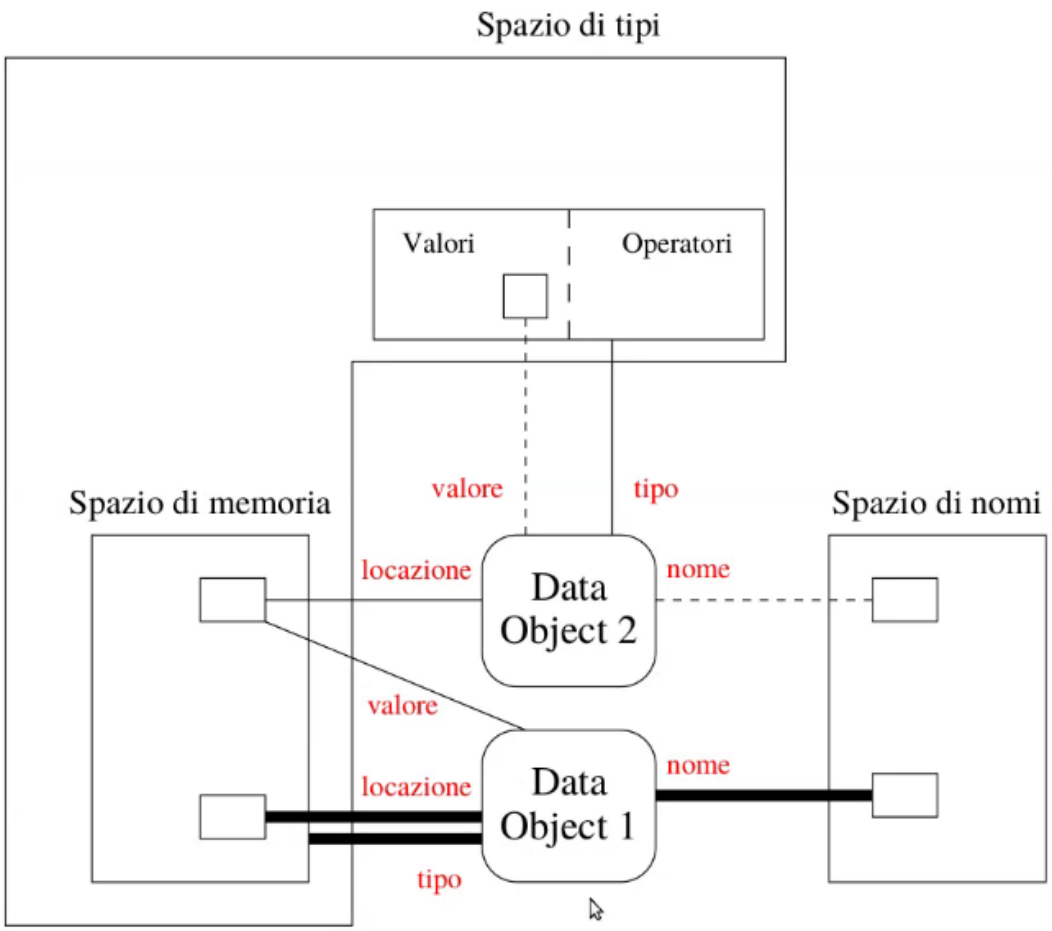
- Durante la compilazione (compile-time);
- Durante il caricamento in memoria (load-time);
- Durante L'esecuzione (run-time).

*Location binding* → avviene o durante il caricamento in memoria o a run-time

*Name binding* → avviene durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione

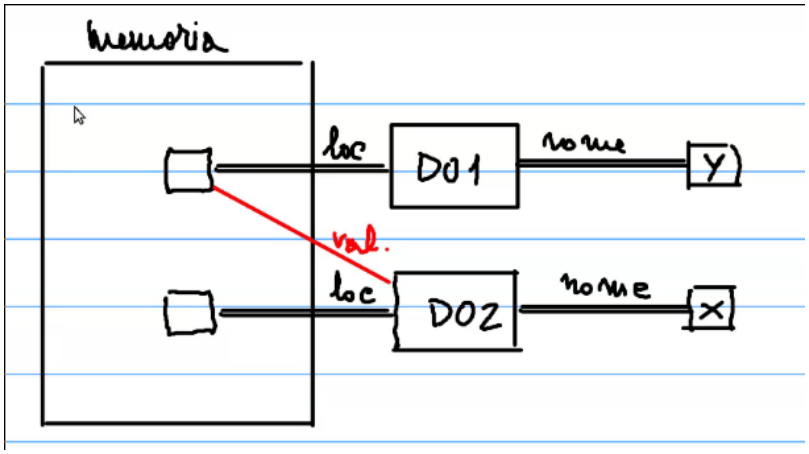
*Type binding* → avviene di solito durante la compilazione (nel momento in cui il compilatore incontra una dichiarazione); un tipo è definito dal sottospazio di valori che un data object puà assumere

1. I puntatori



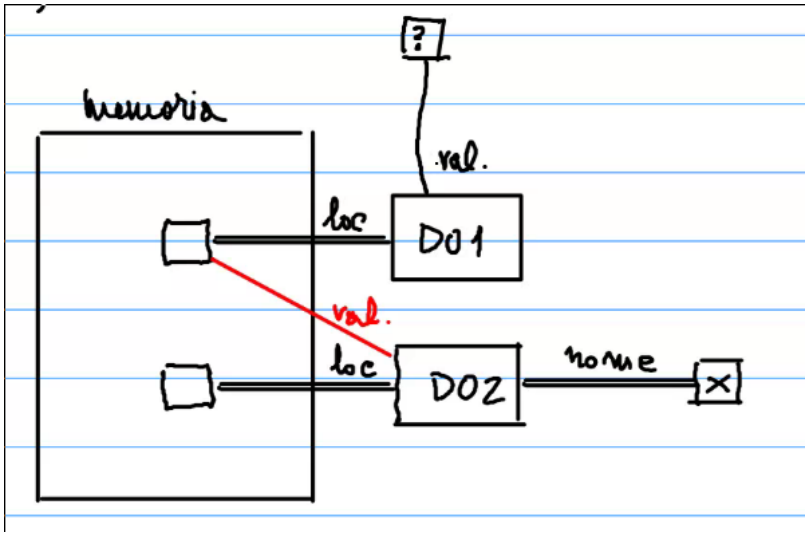

*es*

```
int y, *x; // CREA 2 DATA OBJECT
```




```
x = (int *) malloc(sizeof(int));
```

Il cast (int *) serve a CONTROLLA REC




2 data object coinvolti. Il secondo può non avere un legame di nome o di valore. La deallocazione è necessaria, perchè la modifica del legame di valore genera di solito dati non più accessibili per nome o riferimento

Alcuni linguaggi possiedono meccanismi di recupero automatico di memoria

### 5.1.2 Legame di tipo

Correlato al legame di valore. Ogni volta che il legame viene modificato occorre controllare la consistenza con il legame di tipo.

Un linguaggio è *dinamicamente tipato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza avvengono durante l'esecuzione (*es* linguaggi di scripting e Python).

Un linguaggio è *staticamente tipato* se il legame avviene durante la compilazione; in questo caso il controllo di consistenza può avvenire in entrambe le fasi.

### 5.1.3 Type checking

Mecanismo di controllo di consistenza della coppia valore-tipo. Avviene:

- Durante la compilazione;
- Durante l'esecuzione;

Un linguaggio è *fortemente tipato* se il controllo di consistenza avviene sempre (*es* Java).

Un linguaggio è *debolmente tipato* se il controllo di consistenza può non avvvenire affatto in numerosi casi (*es* C). Un esempio in C sono le union:

```
#include <cstdio>
union numero{
  int numero_piccolo;
  double numero_grande;
} numeri;

int main(){
  numeri.numero_grande = 3.0;
  printf("%d\n", numeri.numero_piccolo);
  printf("%f\n", numeri.numero_grande);
}
```

```
0
3.0
```

Si inserisce un double ma si legge come fosse un int → viene stampato 0 (il compilatore non segnala l'errore).

Essendo esposte a questi tipi di errori si tende a non usarle.

### 5.1.4 Il linguaggio perfetto

Sarebbe conveniente se il type checking avvenisse completamente durante la compilazione e in cui il compilatore non generasse più errori del necessario. Questo linguaggio sarebbe ovviamente fortemente tipato.

Questo linguaggio però non può esistere. Se esistesse il compilatore per essere capace di decidere la correttezza dell'ultima assegnazione in un caso del tipo:

```
int x;
P;
x = " pippo";
```

Dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P (l'istruzione errata eseguita se e solo se la chiamata a P termina).

Pertanto questo linguaggio non sarebbe Turing completo, contro l'ipotesi.

I compilatori nella situazione precedente ignorano se una istruzione viene eseguita o meno e segnalano comunque un errore nell'ultima linea.

```
program wrong (input, output)
var i: integer;
begin
    if false then i:=3.14;
    else i :=0;
end.
```

### 5.1.5 Strategie

La strategia per i linguaggi fortemente e staticamente tipati è:

- Fare più controlli possibili a tempo di esecuzione;
- Eseguire i restanti a tempo di esecuzione.

Prima si scoprono gli errori e più il testing diventa economico.

## 5.2 Blocchi di istruzioni

Istruzioni raggruppate in blocchi per meglio definire:

- Ambito delle strutture di controllo;
- Ambito di una procedura;
- Unità di compilazione separata;
- Ambito dei legami di nome;

I blocchi che definisco l'ambito di validità di un nome contengono due parti:

- Sezione di dichiarazione del nome;
- Sezione che comprende gli enunciati sui quali ha validità il legame.

```
...
BLOCK A;
    DECLARE I;
BEGIN A
...
END A
```

### 5.2.1 Ambito di validità di legami

Essenzialmente due tipi di ambito di validità (scoping):

- ambito statico o lessicale (linguaggi moderni)
  - blocchi annidati vedono e usano i legami dei blocchi più esterni (di solito aggiungon legami locali o sovrapporne di nuovi)
- ambito dinamico (linguaggi come lisp)
  - assume il proprio senso maggiore quando vi sono procedure chiamanti e chiamate (la chiamata vede e usa i legami visti e usati dalla procedura chiamante)

### 5.2.2 Legami di nome

```
PROGRAM P;
    DECLARE X;
BEGIN P
... {X da P}
    BLOCK A;
DECLARE Y;
    BEGIN A
    ... {X da P, Y da A}
BLOCK B;
    DECLARE Z;
BEGIN B
... {X da P, Y da A, Z da B}
END B;
```

```
... {X da P, Y da A}
    END A;
    ... {X da P}
    BLOCK C;
DECLARE Z;
    BEGIN C
    ... {X da P, Z da C}
    END C;
... {X da P}
END P;
```

### 5.2.3 Mascheramento

```
PROGRAM P;
    DECLARE X, Y;
BEGIN P
... {X e Y da P}
    BLOCK A;
DECLARE X, Z;
    BEGIN A
    ... {X e Z da P, Y da P}
    END A;
    ... {X e Y da P}
END P;
```

### 5.2.4 Legami di locazione

Si dice allocazione statica quando le variabili conservano il proprio valore ogni volta che si rientra in un blocco (il legame di locazione è fissato e costante al tempo di caricamento)

```
PROGRAM P;
    DECLARE I;
BEGIN P
    FOR I :=1 TO 10 DO
    BLOCK A;
DECLARE J;
    BEGIN A
IF I=1 THEN
    J:=1; {I da P, J da A}
ELSE
```

```
J:= J*I;
END IF
    END A;
END P;
```

Si dice allocazione dinamica di memoria quando il legame di locazione (e anche di nome) è creato all'inizio dell'esecuzione di un blocco e viene rilasciato a fine esecuzione.

È realizzata attraverso il record di attivazione di un blocco → contiene tutte le informazioni sull'esecuzione del blocco necessarie per riprendere l'esecuzione dopo che essa è stata sospesa.

Il record di attivazione puà contenere informazioni complesse, ma per realizzare un legame dinamico di locazione in blocchi annidati è sufficiente che contenga le locazioni dei dati locali più un puntatore al record di attivazione del blocco immediatamente più esterno

### 5.2.5 Stack di attivazione

In ogni momento dell'esecuzione lo stack di attivazione contiene i record "attivi":

- Il top dello stack contiene sempre il record del blocco correntemente in esecuzione;
- Ogni volta che si entra in un blocco, il record di attivazione del blocco viene posto sullo stack (push);
- ogni volta che si esce da un blocco, viene eliminato il record al top dello stack (pop).

*es*

```
PROGRAM P;
    DECLARE I, J;
    BEGIN P
BLOCK A;
    DECLARE I,K;
BEGIN A
    BLOCK B;
DECLARE I,L;
    BEGIN B
... {I e L da B, K da A, J da P}
    END B;
    ... {I e K da A, J da P}
END A;
BLOCK C
    DECLARE I, N;
```

```
BEGIN C
    ... {I e N da C, J da P}
END C;
... {I e J da P}
END P
```

# 6 Lezione del 19-03

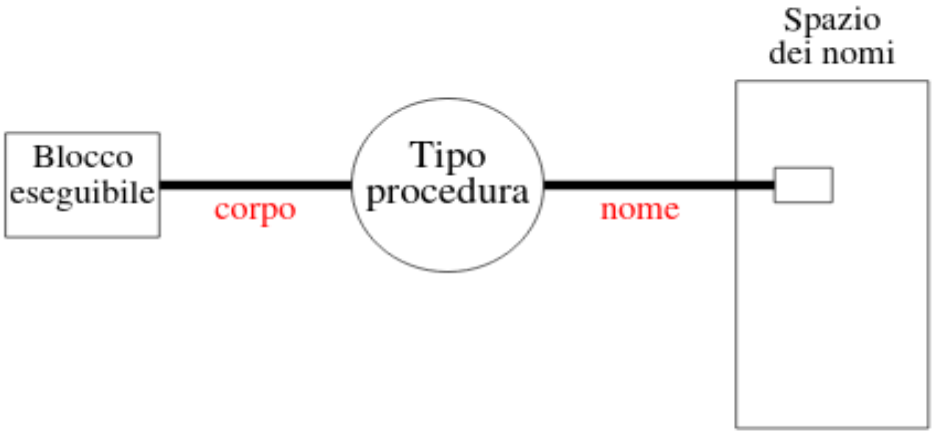
## 6.1 Procedure come astrazioni

Sono astrazioni di parti di programma in unità di esecuzione più piccole, come enunciati o espressioni, in modo da nascondere i dettagli ai fini del loro riuso Vantaggi:

- programmi più facili da scrivere, leggere e modificare;
- Unità di programmi indipendenti o con dipendenze benspecificate a livello più alto;
- Riusabilità del codice.

Se si distinguono come unità di esecuzione, in ordine crescente di complessità, espressioni, enunciati, blocchi, programmi, allora si definisce:

- astrazione procedurale la rappresentazione di una unità diesecuzione attraverso un'altra unità più semplice;
- Rappresentazione di un blocco attraverso un enunciato o un espressione;
- Causa la generazione di un oggetto analogo al Type Object;
- L'invocazione avviene durante l'esecuzione;
- Ogni invocazione diversa della stessa procedura causa lagenerazione di un nuovo "oggetto procedura" con lo stesso legame di tipo, ma con diverso record di attivazione.



## 6.2 Record di attivazione

Il record di attivazione rappresenta l'intero ambiente di esecuzione di una procedura. Esso consiste di solito in:

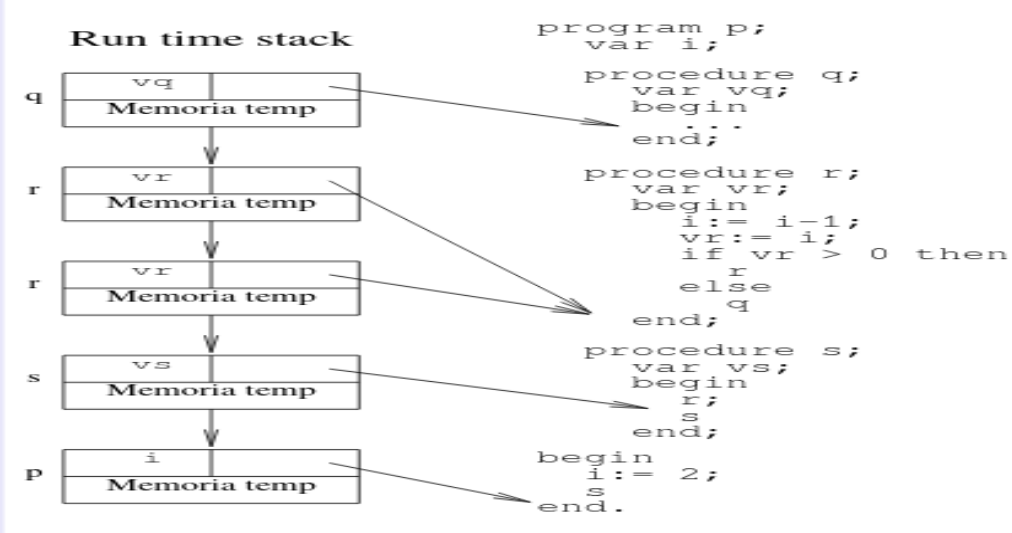
1. ambiente locale (tutti i data object che sono definiti all'internodella procedura);
2. ambiente non locale (tutti i data object la cui definizione'epropagata da altre procedure);
3. ambiente dei parametri (contiene informazioni sui dati che sonopassati [d]alla procedura);

Ogni volta che una procedura viene invocata il suo record di attivazione viene aggiunto al cosiddetto stack di esecuzione. Sul top dello stack c'è sempre il record relativo alla procedura correntemente in esecuzione. Alla terminazione della procedura, il record di attivazione viene rimosso dallo stack.

### 6.2.1 Ambiente locale

Include:

- Tutte le variabili dichiarate localmente;
- Il puntatore alla prossima istruzione;
- Memoria temporanea necessaria alla valutazione delle espressioni contenute nella procedura (altamente dipendente dalla realizzazione)




### 6.2.2 Propagazione dei Data OBJECT

Viene realizzato aggiungendo al record di attivazione un puntatore al record di attivazione della procedura da cui vengono propagate le definizioni o i dati.

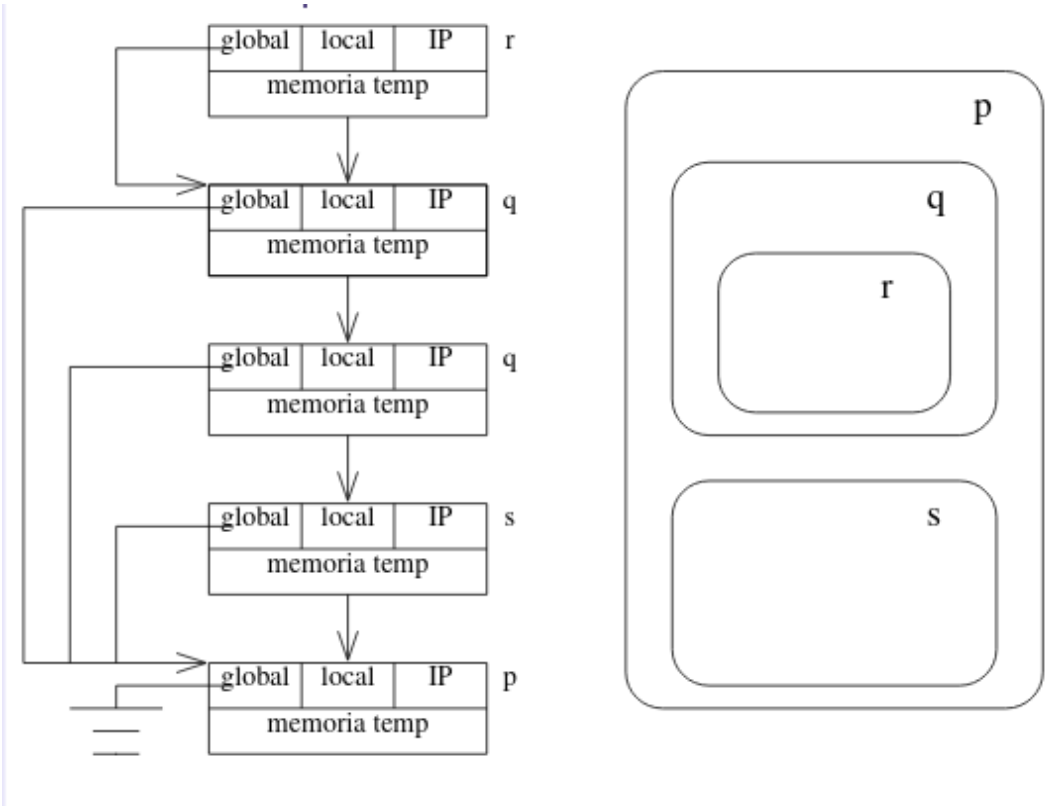
Se viene richiesto l'accesso ad un dato che non è definito localmente esso viene ricercato in modo ricorsivo nei record di attivazione precedenti. Tre tipologie di realizzazioen:

- Propagazione in ambito statico;
  - l'ambiente nonlocale di una procedura è propagato dal programma che la contiene sintatticamente → propagazione di posizione.
- Propagazione in ambito dinamico;
  - l'ambiente nonlocale di una procedura è propagato dal programma chiamante
- Nessuna Propagazione
  - l'uso di ambienti non locali è scoraggiato perchè produce effetti collaterali non facilmente prevedibili

**Propagazione in ambito statico**

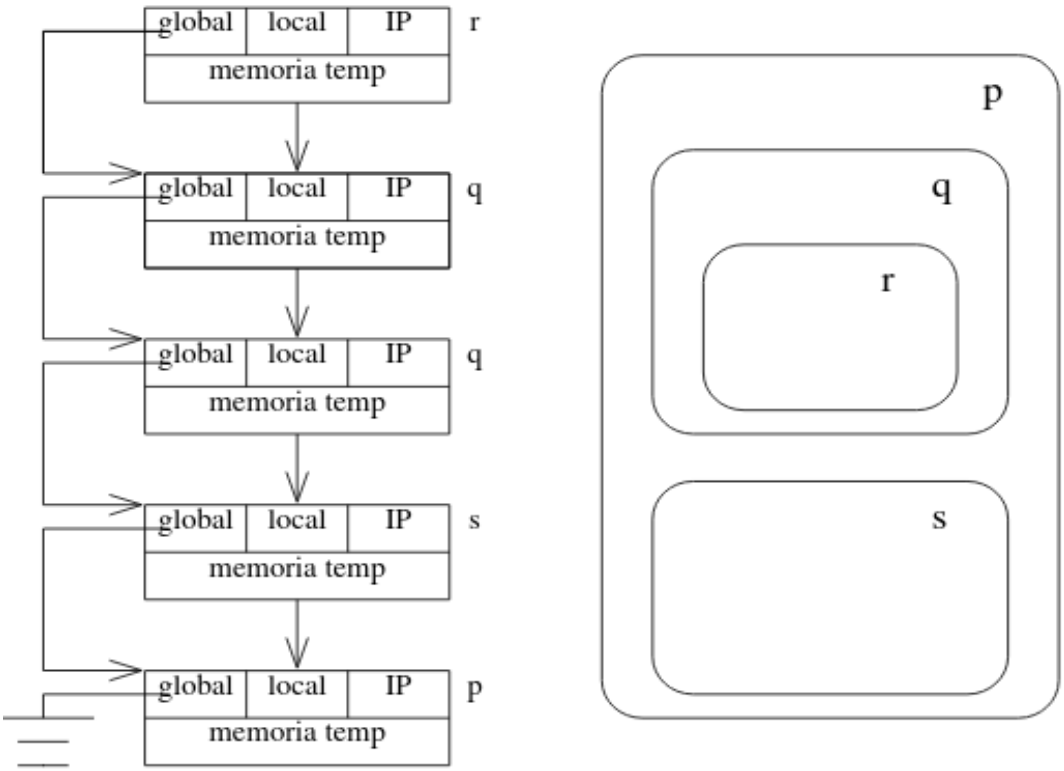
```
program p;
  var a, b, c: integer;

  procedure q;
    var a, c; integer;
    procedure r;
      var a: integer;
      begin {r}          {variabili: a da r; b da p; c da q;
        ...               procedure: q da p; r da q}
      end;  {r}
    begin {q}            {variabili: a da q; b da p; c da q;
       ...               procedure: q ed s da p; r da q}
    end;  {q}

  procedure s;
    var b: integer;
    begin {s}            {variabili: a da p; b da s; c da p;
      ...                procedure: q ed s da p}
    end;  {s}

begin {p}                {variabili: a, b, c da p;
  ...                     procedure: q, s da p}
end.  {p}
```

Supponendo una sequenza di attivazione (p,s,q,q,r), lo stack di esecuzione ha questa forma:




**Ambito dinamico**

La stessa sequenza di attivazione precedente (p,s,q,q,r), generaallora lo stack di esecuzione:




Nella propagazione in ambito dinamico

- Il puntatore all'ambiente non locale non è più necessario;
- È praticamente impossibile la determinazione dell'ambiente di esecuzione di una procedura durante la scrittura del codice sorgente.

### 6.3 Parametrizzazione di procedure

Si distinguono in:

- parametri IN → sono passati dall'unità chiamante all'unità chiamata al momento dell'invocazione;
- parametri OUT → sono passati dall'unità chiamata alla unità chiamante al momento della terminazione della prima;
- parametri IN-OUT → transitare le informazioni in entrambe le direzioni.

Devono essere specificati in due punti:

- definizione della procedura → *parametri formali*;

- invocazione della procedura → *parametri attuali.*

Nella definizione deve essere specificato il tipo dei parametri formali. Nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.

Eccezione comune:

- lasciare i parametri formali senza alcun legame di tipo → il legame si instaura durante l'esecuzione (run time) allo stesso tipo dei parametri attuali (impossibile il type checking in compilazione);
- array a dimensione variabile; il legame di tipo (e l'eventuale controllo di consistenza) verrà realizzato durante l'esecuzione.

Metodi di associazione:

- *per posizione* → a seconda della posizione relativa nella sequenza dei parametri;
- *per nome* → il nome del parametro formale è aggiunto come prefisso al parametro attuale;

*es*

procedure TEST (A: in Atype; b: in out Btype; C: out Ctype) – invocazione che usi associazione per posizione è: TEST(X, Y, Z); – mentre una che usi associazione per nome può essere TEST(A=>X, C=>Z, b=>Y);

Una tecnica è la cosidetta *associazione di default.* Essa permette di specificare valori di default ai parametri formali che non sono stati legati a valori da parametri attuali.

### 6.3.1 Parametri IN

Realizzati in 2 modi:

- Con un riferimento → la locazione del parametro attuale diventa la locazione del parametro formale
  - poichè il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura;
- Con una copia → si alloca il parametro formale all'interno dello stack di attivazione
  - parametro formale e attuale presentano sono allocate in due posizioni diverse;
  - modifica permessa, perchè valida solo nell'ambiente diesecuzione della procedura;

Il secondo modo è meno efficiente del primo, sia rispetto allo spazio sia al tempo, ma è più flessibile e richiede meno variabili locali.

### 6.3.2 Parametri OUT

Realizzati in 2 modi:

1. Con un riferimento;
2. Con una copia.

Rappresentano risultati → alcuni linguaggi assumono che i parametri OUT non siano inizializzati e ne proibiscono la "lettura", ad es:

- Uso a destra di un assegnamento;
- Passaggio a un parametro IN o IN-OUT di un'altra procedura.

### 6.3.3 Parametri IN-OUT

Combinazione dei due precedenti. Realizzati in 2 modi:

- Con un riferimento;
- Con una copia.

## 6.4 Aliasing

È la possibilità di riferirsi alla stessa locazione con nomi diversi. Nel passaggio dei parametri può causare notevoli problemi di interpretazione.

## 6.5 Procedure come parametri di procedure

Alcuni linguaggi permettono l'uso di procedure come argomento di altre procedure.

# 7 Lezione del 23-03

## 7.1 Memo

In ADA sono presenti le keyword *in, out, in out.*

Pascal o non mette nulla (passaggio per copia) o utilizza la keyword *var* (*in out* per riferimento)

C, C++, Java presentano solo il passaggio *in* per copia

In C e C++ si può simulare *in-out* per riferimento (sfruttando gli opportuni operatori)

In Java il passaggio di oggetti somiglia a un passaggio per riferimento (viene passato il puntatore all'oggetto).

## 7.2 Funzioni

Sono procedure che restituiscono un valore alla procedura chiamante. Sono realizzate:

- O creando una pseudovariabile nell'ambiente locale della procedura chiamata
  - questa variabile può essere solo modificata (nessun accesso in lettura);
- O utilizzando una istruzione di return per restituire esplicitamente il controllo alla procedura chiamante inviandole allo stesso tempo il valore di una espressione

# 8 Lezione del 26-03

## 8.1 Esempi sul record di attivazione nei linguaggi

### 8.1.1 Esempio interferenza

```
#include <stdio.h>
void interferenza(int x){
    int var;
    printf("var = %d\n", var);
    var = x;
    }

void interferenza_bis(int x){
    int var;
    printf("var = %d\n", var);
    var = x;
}
```

```
int main (void){
    interferenza(1);
    interferenza_bis(2);
    interferenza(3);
}
```

$$\begin{aligned} \text{var} &= 0 \\ \text{var} &= 1 \\ \text{var} &= 2 \end{aligned}$$

### 8.1.2 Esempio copia

```
#include <stdio.h>
void copy(char x){
  char buf[5];
  int i=0;
  do{
  buf[i] = x[i];
  }
  while(x[i++] != '\0');
}
int main(void){
    char v[] = "abcdef";
    copy(v);
    return 0;
}
```

Si ottiene uno `stack mashing detected` (si cerca di inserire dati in quantità maggiore di quanti ne possa contenere il `buf[5]`)

## 8.2 Macro

Generazione di un nuovo brano di codice sorgente (espansione della macro) in cui i nomi dei parametri attuali sostituiscono i nomi dei parametri formali. *es.* Data la procedura

```
procedure swap (a, b: integer);
var temp: integer;
begin
    temp := a;
    a := b;
```

```
    b := a;
end;
```

La chiamata *swap(x,y)* esegue il seguente brano di codice:

```
temp := x;
x := y;
y := temp;
```

### 8.3 Implemantazione efficiente dello scoping statico

L'accesso ad una variabile non è istantanea → bisogna scandire una lista che potrebbe essere lunga.

### 8.4 Annidamento

```
program p;
var a,b,c int;
    procedure q;
    var a,c int;
procedure r;
var a int;
    procedure s;
    var b int;
```

Una procedura q è annidata in un blocco b se è definita dentro b, ad *es*:

- q e s sono annidate in p;
- r è annidata in q.

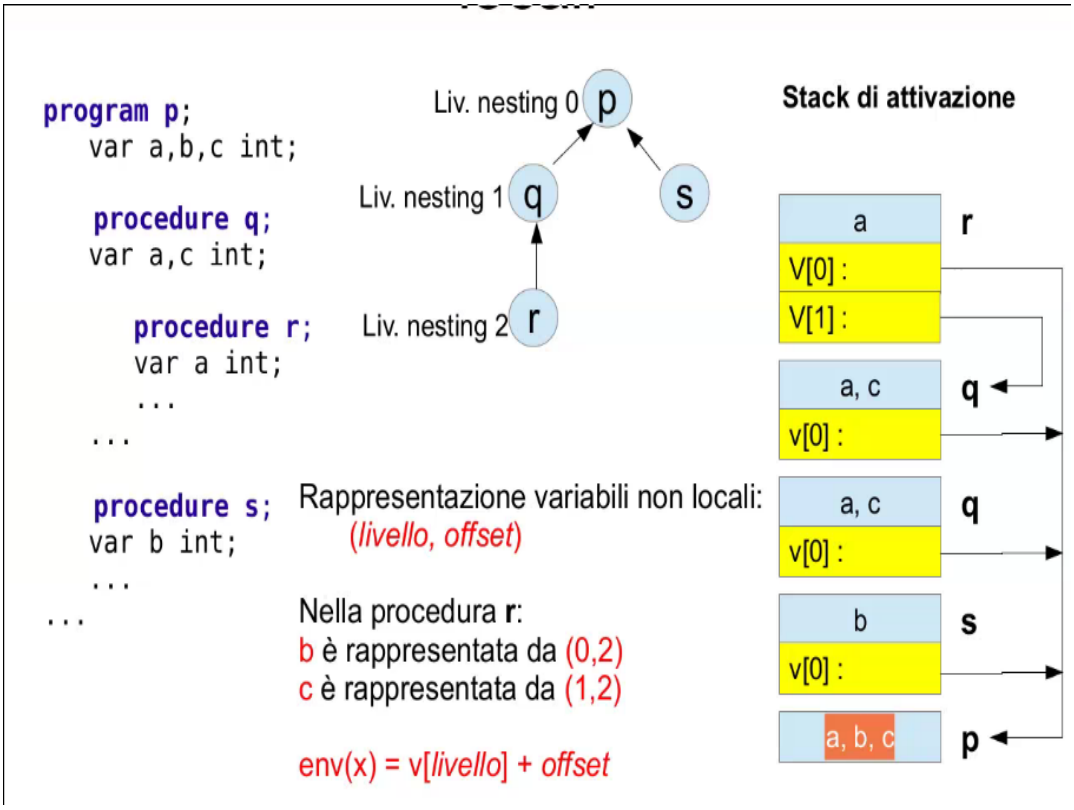
Il livello di nesting di una procedura è il numero di blocchi che a contengono:

- il livello di nesting di $q$ ed $s$ è 1;
- Il livello di nesting di $r$ è 2.

L'ambiente statico non locale di una procedura è dato dall'ambiente delle procedure in cui è innestata

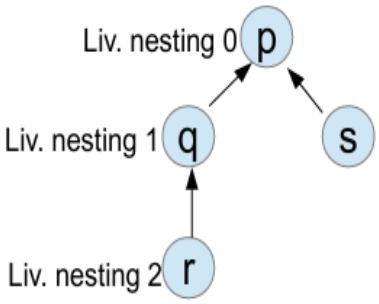
L'annidamento si può rappresentare come un albero. I livelli dell'albero di nesting corrispondono ai livelli di nesting

## 8.5 Rappresentazione variabili non locali



## 8.6 Mantenimento del vettore degli ambienti non locali

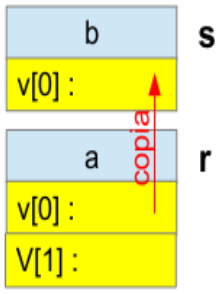
```
program p;
   var a,b,c int;

   procedure q;
   var a,c int;

      procedure r;
      var a int;
      ...
   ...

   procedure s;
   var b int;
   ...
...
```




Chiamata a procedure definite esternamente:

**Esempio 2: r chiama s:**

- il liv. di nesting **decresce**
- Copiare solo gli elementi di livello *minore o uguale* a quello di s
- Struttura ad albero + scoping statico garantiscono sempre di trovare l'ambiente non locale della procedura chiamata nel record del chiamante




## 8.7 Proprietà di questa implementazione

- Accesso alle variabili in tempo costante
  - Accesso a vettore + somma del puntatore ivi contenuto e dell'offset;
  - Indipendente dal livello di nesting;
  - Calcolo supportato direttamente dalle istruzioni macchina.
- Creazione record di attivazione lineare nel livello di nesting (copia degli elementi di v[])
  - Indipendente dall'esecuzione;
  - Dipende solo dal testo del sorgente;
- Tempo costante;
- L'implementazione naive richiederebbe a run time di percorrere le liste di puntatori all'ambiente non locale
  - Accesso alle variabili in tempo **lineare** nel livello di nesting;
  - Creazione dei record di attivazione in tempo **lineare** nel livello di nesting.

## 8.8 Paradigma OO

Rappresentazione di ogni cosa in un programma OO.

In essenza → progettare un sistema OO consiste nell'identificare quali oggetti il sistema debba contenere, il loro comportamento e le responsabilità di ognuno di essi e come essi interagiscono tra loro

## 8.9 Encapsulation

Le strutture dati che implementano i valori sono accessibili direttamente solo dalle operazioni fornite dal tipo → è una regola di **scoping**

## 8.10 C++ vs Java

**C++**

Acquisisce eredità di C ed è compatibile con programma C esistenti.

Cresciuto in modo abnorme ed è veramente difficile raggiungere piena competenza in tutti gli aspetti del linguaggio.

**Java**

WWW e abilità di eseguire *web applet* su qualunque elaboratore e SO.

Linguaggio ben progettato con pochi costrutti. Risolti i prooblemi della velocità di esecuzione.

Diffusamente utilizzato nell'ambito educativo

## 8.11 UML

Linguaggio i cui simboli sono diagrammi

## 8.12 Essenza di un oggetto

- Insieme di **attributi** (valore, stato interno, o qualunque altra cosa sia necessaria per il modello dell'oggetto);
- **Abilità** di modificare lo stato degli attributi;
- **Responsabilità** sotto forma di servizi offerti ad altri oggetti.

  Gli oggetti esterni non hanno conoscenza di come un oggetto realizza gli attributi internamente, ma piuttosto devono conoscere quali servizi un oggetto offre.

### 8.13 Classe

Descrizione di un insieme di oggetti che condivdono attributi e comportamento comuni. Simile a quello del tipo di dato astratto nei linguaggi non OO

La definizione di una classe descrive tutti gli attributi comuni agli oggetti della classe, così come tutti i metodi che realizzano il comportamento comune degli oggetti della classe.

Gli oggetti membri di una classe sono chiamati **istanze** di quella classe → I membri di una classe variano durante l'esecuzione di un programma

### 8.14 Polimorfismo

Permette al mittente di ignorare l'identità (il tipo) del ricevente.

Realizzato con un mix di overriding e upcasting.

## 9 Lezione del 30-03

### 9.1 Classi astratte

Definiscono una interfsaccia comune a tutte le sottoclassi, specificando solo i nomi (la firma) dei metodi che le sottoclassi "concrete" dovranno definire.

Poichè sono incomplete della realizzazione dei metodi, le classi astratte non possono avere istanze.

Una classe che estende una classe astratta eredita tutti i suoi metodi (inclusi quelli astratti). Ogni classe che eredita metodi astratti è considerata astratta essa stessa, a meno che essa nonrealizzi tutti i metodi astratti ereditati

### 9.2 Visibilità delle componenti di una classe

È la possibilità di una classe di veere ed usare le risorse di un'altra classe. Livello di visibilità:

- private (`-`) → attributi e metodi sono visibili solo a oggetti istanze della stessa classe;
- Package (`#`) → attributi e metodi sono visibili ad un insieme specifico di classi;
- Public (`+`) → attributi e metodi sono visibili a tutti gli altri oggetti.

Normalmente si cerca di evitare attributi publici. Se si deve reperire o modificare le informazioni sullo stato interno di uno oggetto, lo si fa utilizzando i metodi specifici (e visibili) dell'oggetto

## 9.3 Ereditarietà

Il concetto di ereditarietà descrive come si possano condividere attributi e funzionalità tra classi di natura o scopo simile.

Vi sono due modi con cui si può aggiungere il concetto di ereditarietà al modello del sistema:

- Generalizzazione;
- Specializzazione.

### 9.3.1 Generalizzazione

Avviene quando si riconosce che più classi dello stesso diagramma Class esibiscono funzionalità,struttura, scopo comuni.L'analista decide allora di creare una nuova classe contenente gli attributi e le funzionalità comuni esemplifica le classi precedenti come estensioni della nuova(generalizzata).

### 9.3.2 Specializzazione

Avviene quando si riconosce che la nuova classe che si sta aggiungendo ha le stesse funzionalità,struttura e scopo di una classe esistente, ma ha bisogno di nuovo codice o attributi.La nuova classe è una forma *specializzata* di quella già esistente.

## 9.4 Associazioni e molteplicità

### 9.4.1 Associazioni complesse

Sono quelle associazioni in cui vi sono molteplicità maggiori di 1 in entrambe i ruoli. Difficile da implementare. 2 possibili soluzioni:

- Classe di associazione
  - Il nome della classe deve essere lo stesso della associazione;
  - Una classe di Associazione è usata per documentare la risoluzione di una associazione complessa, non fa parte delle astrazioni chiave;
  - Restano nei diagrammi Class finchè, nella fase di progettazione, non si decide di realizzarle;
- Associazione qualificata
  - Usate quando si vuole evidenziare che la realizzazione futura sarà mediante un array, una hash table, un dizionario;

### 9.4.2 Ruoli nelle associazioni

- Riferiscono ad una associazione tra classi;

- Etichette poste alle estremità di una associazione;
- Rappresentano un attributo all'interno della classe che si trova all'estremità opposta del nome.

### 9.4.3 Associazioni riflessive

In un diagramma Object possono esistere link tra istanze della stessa classe, ma c'è una sola classe nel diagramma Class per rappresentare entrambi gli oggetti. In tal caso è usata una associazione riflessiva.

## 9.5 Introduzione a Java

Un linguaggio di programmazione, con sintassi simile a C++ e semantica simile a SmallTalk.

Le applicazioni Java sono programmi autonomi che richiedono un elaboratore su cui sia installato Java Runtime Environment(JRE).

Un ambiente di sviluppo, cioè un insieme di numerosi strumenti per la realizzazione di programmi: un compilatore, un interprete, un generatore di documentazione, uno strumento di aggregazione di file, etc...

### 9.5.1 Errori comuni

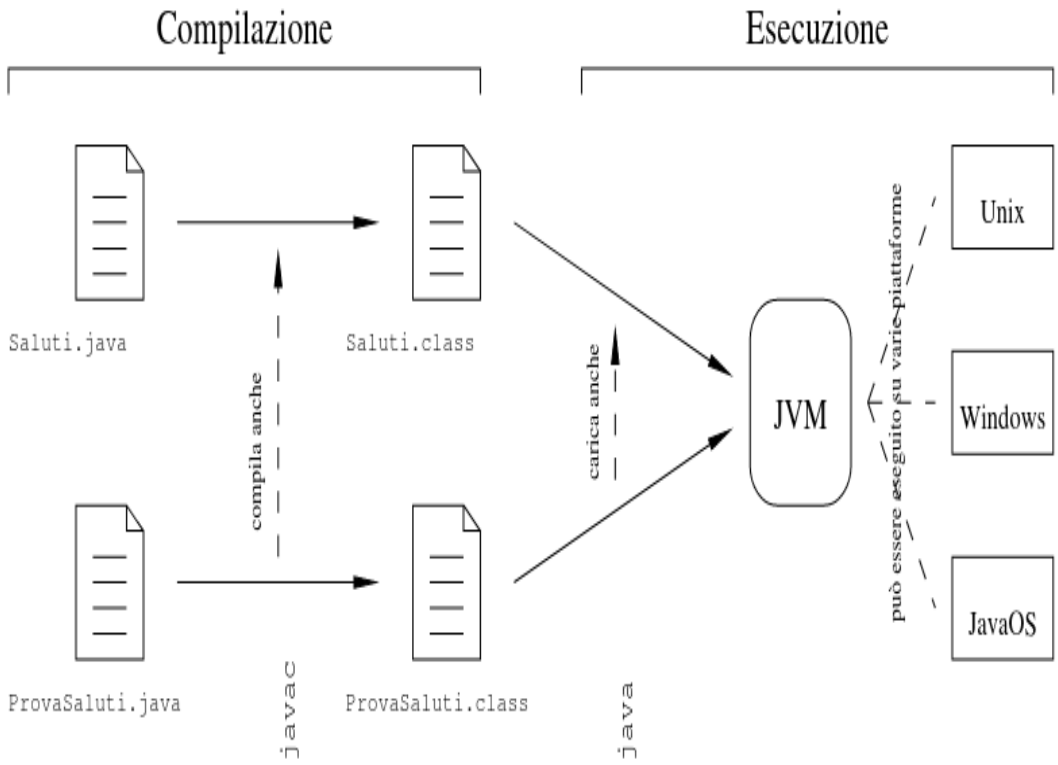
Errori in compilazione:

- Installazione JavaSDK non corretta;
- Nome di metodi di sistema non corretti;
- Errata corrispondenza tra nome di classe e nome di file;
- Numero di classi pubbliche in un file;

Errori in esecuzione:

- Errata scrittura della classe da caricare inizialmente in memoria;
- Errata scrittura della firma del metodo main;

### 9.5.2 Compilazione ed Esecuzione




### 9.5.3 Sicurezza

Miglioramento della sicurezza del codice che proviene da fonti esterne.

Il Class Loader:

- Carica solo le classi necessarie per l'esecuzione del programma;
- Mantiene le classi del file system locale in uno spazio di nomi separato
  - limita applicazioni "cavallo di Troia", poichè le classi locali sono sempre caricate per prime;
- previene il così detto *spoofing*.

Il Bytecode Verifier controlla che:

- Il codice rispetti le specifiche della JVM;
- Il codice non violi l'integrità del sistema;
- Il codice non generi stack overflow o underflow;
- Non vi siano conversioni di tipo illegali (*es* la conversione di interi in puntatori).

### 9.5.4 Costruttori

**NB**:

- I costruttori NON sono metodi;
- I costruttoni NON sono ereditati;
- Un costruttore NON ha valore di ritorno;
- Gli unici modificatori validi per il costruttore sono public,protected e private.

**Costruttore di default**

C'è sempre almeno un costruttore in ogni classe. Se il programmatore non scrive nessun costruttore, allora verrà usato un costruttore di default.

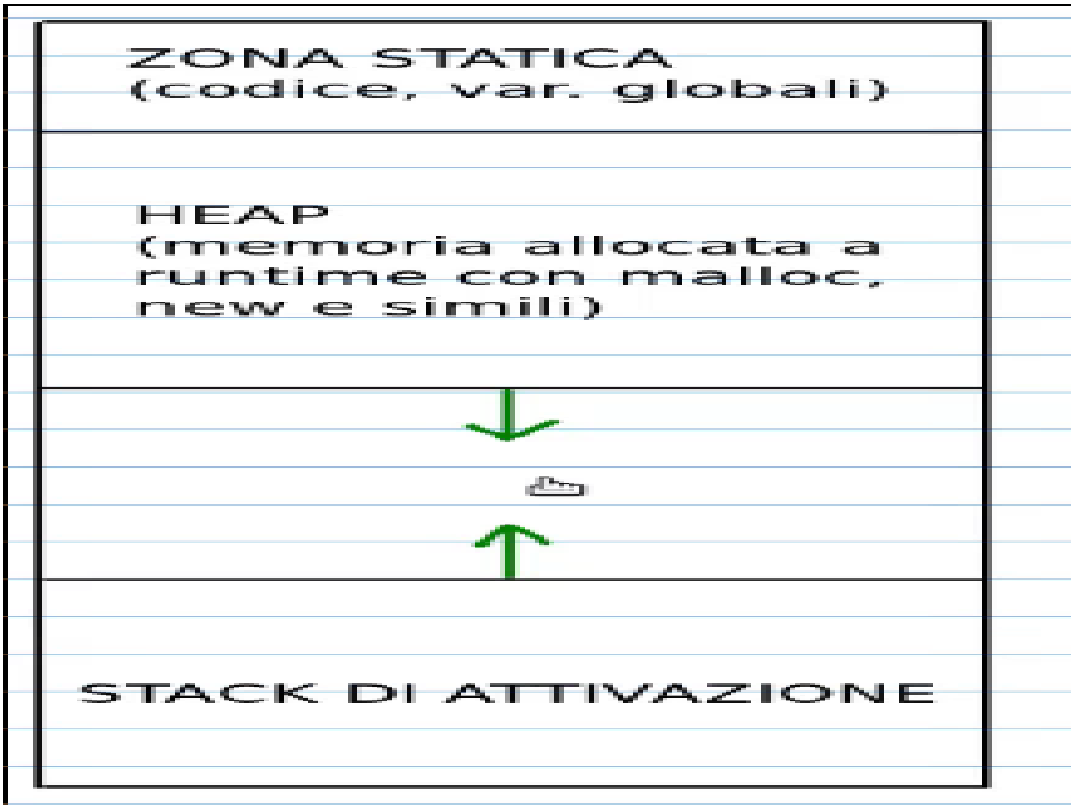
Il costruttore di default non ha argomenti, non ha corpo.

Permette di creare istanze di classi (con l'espressionenew Xxx()) senza dover scrivere un costruttore.

**NB** → Se si aggiunge una dichiarazione di costruttore con argomenti ad una classe che in precedenza non aveva costruttori espliciti, allora si perde il costruttore di default. Da quel punto in poi, una chiamata a *new Xxx()* genererà un errore di compilazione.

# 10 Lezione del 09-04

## 10.1 Organizzazione della memoria nei linguaggi di programmazione




## 10.2 Ambiente non locale nel paradigma a oggetti

- Consiste negli attributi dell'oggetto che esegue il metodo;
- È ancora una forma di scoping statico
  - Attributi dichiarati nel blocco di classe che contiene il metodo;
- Allocato nello heap;
- Viene indicato esplicitamente
  - con il destinatario del messaggio;
  - *this* se il destinatario non viene specificato.

## 10.3 Java

### 10.3.1 Commenti

Stessa logica del `C/C++`.

### 10.3.2 Blocchi

Suddivisi o su più righe o racchiuse tra parentesi

### 10.3.3 Identificatori

- Sono nomi assegnati a variabili, classi e metodi;
- Possono cominciare con un carattere Unicode, underscore, o il dollaro;
- Sono case sensitive e non hanno lunghezza massima;
- Il nome di una classe è costituito solo da caratteri ASCII per mantenere la massima portabilità.

### 10.3.4 Tipi primitivi

- Logici → boolean (true e false);
- Testuali → char;
- Interi → byte, short, int, long;
- Floating point → float, double.

Non c'è cast tra tipi interi e boolean. Interpretare valori numerici come valori logici non è consentito in java.

### 10.3.5 Tipo String

Il tipo String non è un tipo primitivo, ma una classe (è scritto infatti con la S maiuscola). Ha i suoi litterali compresi tra apici doppi

### 10.3.6 Tipi Reference

Tutti i tipi primitivi sono tipi *reference*. Una variabile reference contiene la "maniglia" di un oggetto

### 10.3.7 Costruzione degli oggetti

`new Xxx()` serve ad allocare spazio per il nuovo oggetto. Scatena le seguenti operazioni:

- Viene allocato lo spazio per il nuovo oggetto e le variabili dell'istanza sono inizializzati al loro valore di default;
- Viene eseguita ogni inizializzazione esplicita degli attributi;
- Viene eseguito un costruttore;
- Viene assegnato il riferimento finale dell'oggetto.

### 10.3.8 Parametri

- Java permette il passaggio dei parametri per valori (parametri IN realizzati per copia);
- Il passaggio per riferimento (che permette la modifica del valore del parametro nel contesto della procedura chiamante) è proibito in Java;
- Quando si passa una istanza di oggetto come argomento di un metodo, quello che si sta passando non è l'oggetto, ma solo un riferimento a quell'oggetto
  - Questo riferimento è copiato nel parametro formale;
- Sarà possibile la modifica nel consteso del chiamante (mai del valore della variabile) dell'oggetto a cui fa riferimento quello variabile.

### 10.3.9 Il riferimento this

La parola chiave può essere usata:

- Per fare riferimento, all'interno di un metodo o di un costruttore locale, ad attributi o metodi locali
  - Questa tecnica è usata per risolvere ambiguità in alcuni casi in cui una variabile locale di un metodo maschera un attributo locale dell'ggetto;
- Per permettere ad un oggetto di passare il riferimento a se stesso come parametro ad una altro metodo o costruttore.

### 10.3.10 Convenzioni sul codice

- Package → package oggetti.*;
- Classi → class NomeClasse;
- Interfaccia → interface NomeInterfaccia;
- Metodi → getNomeMetodo();
- Variabli → dimensioniX;
- Costanti → `PI_GRECO`;
- Spaziatura →
  - un solo enunciato per riga;
  - due spazi di indenting per blocchi annidati.

## 10.4 Conversioni

All'inizio esistevano solo i tipi elementari e nessuna gerarchia di classi. Vengono in seguito introdotti i tipi user-defined.

### 10.4.1 Type equivalence

- Name equivalence → i tipi di $x$ e $y$ devono avere lo stesso nome (stesso tipo);
- Structural equivalence → i tipi di $x$ e $y$ devono avere la stessa rappresentazione interna
  - Più snello e flessibile ma più soggetto ad errori.

In C si punta sempre al secondo caso, tranne per le struct dove è richiesto il Name equivalence. Questo scelta deriva dalla richiesta di verifica di una proprietà detta *bisimulazione*.

```
Struct S1{int a; struct S1* next;}; Struct S2{int a; struct S3* next;}; Struct S3{int a; struct S2* next;};
```

- Caso semplice → sia S1 che S2 generano tutte le catene come quelle a destra;
  - Complicazioni in presenza di record varianti

### 10.4.2 Compatibilità di tipi

Con l'avvento dei linguaggi OO l'equivalenza viene rimpiazzata da *compatibilità*. Qualunque sottotipo di T è compatibile con T.

Nei linguaggi OO pi comuni la compatibilità è basata su *nome* piuttosto che *struttura*:

- Due classi con nomi diversi sono diverse;
- Anche se hanno gli stessi attributi e gli stessi metodi;
- Inoltre una classe per essere sottotipi di un'altra deve essere esplicitamente dichiarata tale (*extends*)
  - La struttura non conta neanche in questo caso

## 10.5 Variabili

Variabili definite all'interno di un metodo sono dette **locali** → create quando il metodo è invocato e distrutte alla sua terminazione.

Esse devono essere inizializzate esplicitamente prima di essere usate.

Variabili usate come parametro di metodi, visto il meccanismo di passaggio di parametri (IN per copia) sono variabili locali.

Variabili definite all'esterno di un metodo sono creat quand viene costruito un oggetto. Di 2 tipi:

- **Variabili di classe** → dichiarate usando il modificatore *static*

  - Create nel momento in cui la classe è caricata in memoria;
  - Continuano ad esistere finchè la classe resta in memoria;
  - Una variabile valida per tutte le istanze;

- **Variabili di istanza** → dichiarate senza usare il modificatore *static*

  - Create durante la costruzione dell'istanza;
  - Esistono finchè l'oggetto esiste

### 10.6 Inizializzazione

Nessuna variabile può essere usata prima di essere inizializzata. Le variabili non locali sono inizializzate dalla JVM, nel momento in cuo la classe è caricata in memoria o in cui è allocato spazio per il nuovo oggetto ai seguenti valori:

| Variabile | Valore |
|---|---|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0F |
| double | 0.0D |
| char | '̆000' |
| boolean | false |
| riferimenti | null |

Le variabili locali (quelle dei metodi) devono essere inizializzate manualmente prima dell'uso.

## 11 Lezione del 13-04

### 11.1 Operatori in Java

Simili al `C/C++`.

#### 11.1.1 Booleani

Operatori booleani → `!  (NOT), & (AND), | (OR), ^(XOR)`; Operatori booleani con corto circuito → `|| (OR), && (AND)`

#### 11.1.2 Di bit

Operatori di manipolazione di bit su interi `~ (Complemento a 1), & (AND), | (OR), ^(XOR)`

### 11.1.3 Right e Left Shift

Nello shift a destra aritmetico o con segno (»`) il bit di segno viene conservato. Nello shift a destra logico o senza segno (»>`) il bit di segno è azzerato nello spostamento. Se l'operando di sinistra è un `int`, prima di applicare lo shift viene ridotto l'operando di destra modulo 32; se l'operando è un `long` prima di applicare lo shift viene ridotto l'operando di destra modulo 64

x »> 64 non modifica il valore di x

L'operatore »> è ammesso solo per i tipi interi e long. Se viene usato per `short` o `byte` questi valori vengono promissi con estensione di segno, ad `int` prima di aplicare lo shift.

Per lo shift a sinistra non c'è il problema del bit di segno

### 11.1.4 Concatenazione di stringhe

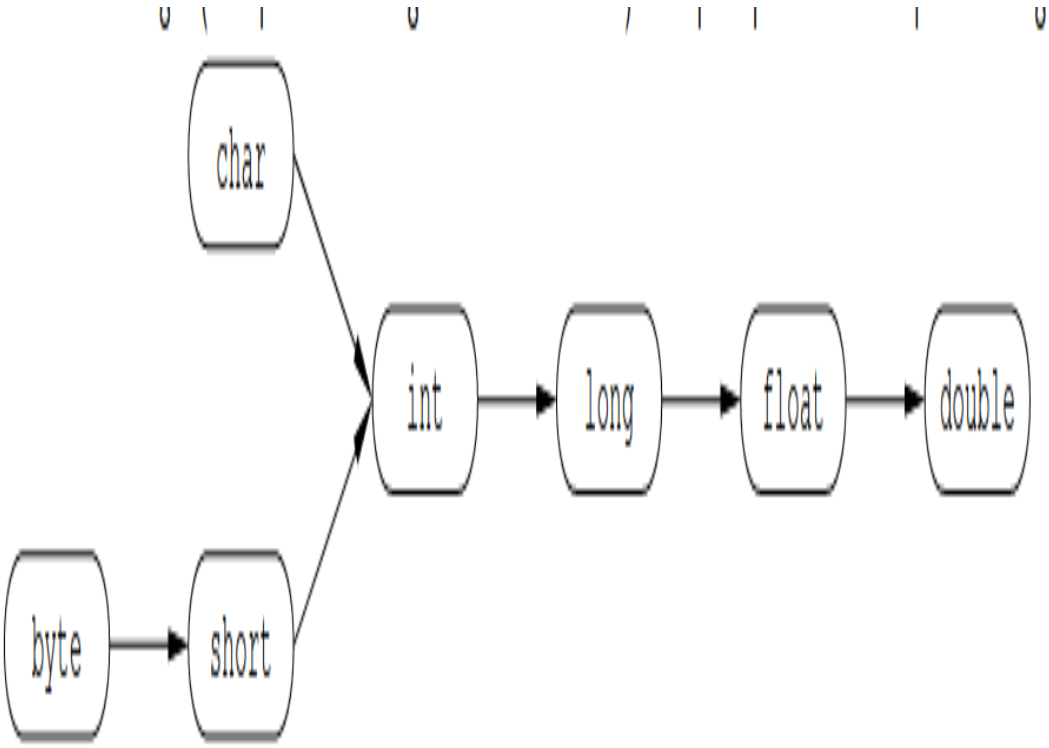
L'operatore + esegue la concatenazione di 2 oggetti di tipo String, producendo un nuovo oggetto di tipo String. Un argomento deve essere un oggetto String. Gli altri argomenti sono convertiti a String automaticamente, attraverso il metodo `toString`.

## 11.2 Casting di primitivi

Una conversione di tipo per assegnazione avviene quando si assegna un valore ad una variabile di tipo differente. Ciò può avvenire o con un operatore di assegnazione, o durante il passaggio di parametri ad un metodo.

Il tipo booleano non può mai essere convertito

### 11.2.1 Conversioni legali




### 11.2.2 Conversioni esplicite

Se è possibile la perdita di informazioni in una assegnazione o in un passaggio di parametri, il programmatore DEVE confermare l'assegnazione con un *cast* esplicito. In un cast esplicito, il valore in eccesso viene troncato.

Ad esempio l'assegnazione da `long` a `int` richiede un cast esplicito

Il casting esplicito non è necessario per i litterali interi (non i floating point) che ricadono nel range legale del tipo di destinazione.

### 11.2.3 Promozione aritmetica

In una espressione aritmetica, le variabili sono automaticamente promosse ad una forma più estesa per non causare perdita di informazioni.

Le regole per operatori unari:

- Se l'operando è un `byte`, uno `short`, un `char` esso è convertito ad `int`
  - (a meno che l'operatore non sia `++` o `--`, nel qual caso non avviene nessuna conversione);

– Altrimenti nessuna conversione;

Le regole per operatori binari:

- Se uno degli operandi è un `double`, allora l'altro operando è convertito ad un `double`;
- Se uno degli operandi è un `float` allora l'altro operando è convertito ad un `float`;
- Se uno degli operandi è un `~long` allora l'altro operando è convertito ad un `long`;
- Altrimenti, entrambi gli operandi sono convertiti ad `int`.

## 11.3 Enunciati Branch

### 11.3.1 If & else

Richiede una espressione booleana, non un intero come in `C/C++`

### 11.3.2 Switch

```
switch  (<expr >){
   case <constant1 >:
     <statement >*
       break;
   case <constant2 >:
     <statement >*
       break;
   default :<statement >*
      break;
```

Dove:

- `<expr>` → deve essere compatibile per assegnazione con `int` (char, byte, short sono promossi automaticamente);
- l'etichetta opzionale `<default>` è usata per specificare il segmento di codice da eseguire quando il valore `<expr>` non corrisponde a nessuno dei valori delle stanze case;
- Se non è presente l'enunciato opzionale `break`, l'esecuzione continua nella stanza case successiva.

## 11.4 Enunciati loop

### 11.4.1 For

```
for (<init_expr >; <bool_expr >; <alt_expr >)
 {<statement >*}
```

### 11.4.2 While

```
while (<bool_expr >) {
<statement >*}
```

### 11.4.3 do/while

```
do {<statement >*
} while (<bool_expr >);
```

## 11.5 Controlli speciali

- `break [label]` → usata per uscire in modo prematuro da un enunciato `switch`, da enunciati di loop o da blocchi etichettati;
- `continue [label]` → usata per saltare direttamente alla fine del corpo di un loop;
- `label: <statement>` → usata per identificare un qualinque enunciato verso il quale può essere trasferito il controllo.

# 12 Lezione del 16-04

## 12.1 Array

Si possono dichiarare array di tipi primitivi o di riferimenti ad oggetti.

Un array è un oggetto → le dichiarazioni precedenti creano solo il riferimento al rispettivo oggetto.

Quindi:

- Non viene qui generato l'oggetto → la generazione richiede sempre l'uso della new;
- Nella dichiarazione non si deve specificare la dimensione dell'array;
- Se si dichiarano più array di uno stesso enunciato usando le parentesi quadre a sinistra, le parentesi sono applicate a tutte le variabili alla loro destra

`int a[], b` → a è riferimento ad array, b è int;

`int [] a, b` → $a$ e $b$ entrambi riferimenti ad array

### 12.1.1 Creazione

Per la creazione dell'array si usa sempre la `new`.

### 12.1.2 Inizializzazione

Poichè l'inizializzazione delle variabili è cruciale, Java offre 2 metodi abbreviati per gli array.

Il primo consiste nella dichiarazione, costruzione ed inizializzazione in una riga

```
String [] nomi = {
    "Antonio",
    "Marcello",
    "Anna"
}
// equivalente a
String [] nomi;
nomi = new String[3];
nomi[0] = "Antonio";
nomi[1] = "Marcello";
nomi[2] = "Anna";
```

Il secondo metodo è la costruzione ed inizializzazione di un array anonimo:

```
int [] esempio1;
esempio1 = new int[] {4,7,3};

// O ancora meglio
  public class A{

      void prendiArray(int [] unArray){
// usa unArray
      }
      public static void main(String[] args){
A a = new A();
a.prendiArray(new int[] {3,4,5,6,7});
      }
  }
```

In java tutti i parametri dei metodi sono di tipo IN (realizzati) per copia. Questo significa che il valore del parametro nel metodo chiamante non può essere modificato. Tuttavia, è possibile creare un riferimento a tale parametro e realizzare ugualmente la modifica nel metodo chiamante.

### 12.1.3 Multidimensioni

- Array di array;
- Array non rettangolare di array;
- Array rettangolare di array.

Costruzione e inizializzazione → `int [][] multiD = {{5,4,3,2}, {9,8}, {7,6,5}};`

### 12.1.4 Estremi

- Indice iniziale 0;
- Numero di elementi è parte dell'array, nell'attributo lenght;
- Accesso oltre i limiti causa il lancio di una eccezione a runtime.

### 12.1.5 Assegnazione

In una assegnazione l'array deve avere lo stesso numero di dimensioni della variabile di riferimento.

### 12.1.6 Ridimensionamento

Gli array non sono ridimensionabili. Si può usare la stessa variabile per riferirsi ad un nuovo array.

### 12.1.7 Copia

Si utilizza il metodo `System.arraycopy()`.

Questo metodo copia i valori contenuti negli elementi dell'array → nel caso di array di oggetti (o di array multidimensionali), ciò significa che vengono copiati i riferimenti agli oggetti, non vengono cioè create nuove copie di oggetti.

### 12.1.8 Aiutare il GC

Il garbage collector non è onnipotente → necessita a volte di un piccolo aiuto. Infati se l'utilizzatore del metodo abbandona il valore di ritorno ricevuto, esso non sarà eleggibile per GC, finchè il riferimento ad esso non sarà sovrascritto. Questo potrebbe richiedere tempi lunghissimi.

## 12.2 Ereditarietà

### 12.2.1 Specializzazione

- Una sottoclasse eredita tutti i metodi e le variabili della superclasse;
- Una sottoclasse non eredita i costruttori della superclasse;

- I 2 modi (esclusivi uno con l'altro) per includere i costruttori in una classe sono
  - Usare il costruttore di default (senza parametri);
  - Scrivere 1 o più costruttori espliciti.

## 12.3 Polimorfismo

- Un oggetto ha una sola forma;
- È l'abilità di riferirsi con la medesima variabile ad oggetti con forme diverse (credendo, però di riferirsi ad un oggetto di una particolare forma ed ignorando, cioè la reale forma dell'oggetto);

## 12.4 Collezioni eterogenee

### 12.4.1 Collezioni omogenee di oggetti dela stessa classe

```
MiaData[] date = new MiaData[2];
date[0] = new MiaData(5,5,2006);
date[1] = new MiaData(25,12,2006);
```

### 12.4.2 Collezioni eterogenee di oggetti di classi diverse:

```
Impiegato[] staff = new Impiegato[100];
staff[0]  = new Impiegato();
staff[1]  = new Impiegato();
staff[2]  = new Quadro();
```

## 12.5 Argomenti polimorfici

- Si possonno costruire metodi che accettino come parametri un riferimento "generico" ed operare in modo automatico su un più vasto insieme di oggetti;
- In questo caso è il metodo ad ignorare la real natura (forma) dell'oggetto che gli viene specificato come parametro attuale;

## 12.6 Operatore `instanceof`

Visto che è lecito scambiarsi oggetti usando riferimenti ai loro antenati, potrebbe essere necessario sapere esattamente la forma dell'oggetto con cui si ha a che fare.

*es*

```
public class Impiegato extends Object // Ridondante
```

```
public class Quadro extends Impiegato
public class Segretario extends Impiegato
```

Se si riceve un oggetto usando un riferimento ad *Impiegato*, esso potrebbe essere anche realmente un *Quadro* o un *Segretario*

## 12.7 Casting

- Nel caso in cui si riceva un riferimento ad un oggetto e che (usando `instanceof`) si sappia che l'oggetto è realmente di una sottoclasse, si dovrebbe comunque usare quell'oggetto come se fosse della superclasse;
- La formalizzazione di polimorfismo rende invisibili tutte le specializzazioni proprie di quell'oggetto;
- Per redenere visibili tutte le caratteristiche dell'oggetto è necessario un *cast* esplicito;
- Se non ci fosse il cast, il metodo risulterebbe inaccessibile al compilatore;

Ogni tentativo di effettuare un cast di riferimenti ad oggetti è soggetto a regole precise.

Siano $A$,$B$,$C$ 3 tipi:

- Gli upcasting sono sempre permessi, e infatti, non richiedono l'operatore di cast (vengono realizzati con una semplice assegnazione);
- Per i downcast il compilatore controlla che l'operazione sia almeno possibile $\rightarrow$ la classe del riferimento di destinazione deve essere una sottoclasse del riferimento di origine
  - Una volta che il compilatore abbia ammesso l'operazione, essa verrà controllata a runtime, per verificare che il tipo dell'oggetto riferito sia compatibile con il tipo di riferimento di destinazione;

## 12.8 Composizioni

- Implica una relazione a vita;
- Se il tutto è creato, anche le parti sono create;
- Quando il tutto muore, anche le parti muoiono;
- Il costruttore per il tutto deve costruire anche le parti.

## 12.9 Aggregazioni

- Nella aggregazione le parti sono create altrove
  - Esse sono inglobate nell'aggregante nel momento in cui esso viene costruito;
- Esiste un loro riferimento anche al di fuori dell'aggregante
  - in questo modo esse non cessano (forse) di esistere dopo la morte di esse

## 12.10 Associazioni

Nella semplice associazioni e vite degli oggetti, sia pure correlate, esistono in modo totalmente indipendente

## 12.11 Overloading e Overriding

Metodi con lo stesso nome, nella stessa classe, che svolgono lo stesso compito con diversi argomenti.

La lista degli argomenti deve essere diversa, perchè essa sola è usata come discriminante.

Il tipo di ritorno può essere diverso, ma non come discriminante per i metodi.

Per i costruttori il discorso è lo stesso dei metodi; il riferimento `this` può essere usato, nella prima linea di un costruttore, per invocarne un altro

Una sottoclasse può sovrappore un metodo ereditato e visibile, a patto che, rispetto al metodo della superclasse, il nuovo metodo abbia:

1. Lo stesso nome, tipo di ritorno e lista di argomenti;
2. Visibilità non inferiore

### 12.11.1 La parola chiave `super`

Viene utilizzata in una classe per riferirsi alla superclasse.

È usata per riferirsi ai membri della superclasse (attributi e metodi) con la dot notation.

I membri della superclasse accessibili con `super` sono ache quelli che la superclasse ha ereditato e non solo quelli esplicitamente definiti in essa.

Non si può usare `super.super.membro` per avvedere a membri di classi di livello superiore alla prima superclasse.

1. Invocazione di un `super-costruttore`

   Si può invocare uno dei costruttori della superclasse usando `super(...)` come prima linea nel proprio costruttore, con analoga sintassi di quella del proprio costruttore.

   Se come prima linea nel proprio costruttore non c'è nè `this` nè `super`, il compilatore pone una chiamata implicita a `super()`, cioè al costruttore di default della superclasse → in questo caso, se non esiste il costruttore di default nella superclasse, il compilatore genera un errore.

## 12.12 Costruzione ed inizializzazione di oggetti

- Viene allocata nello heap la memoria per il nuovo oggetto e sono inizializzate le variabili di istanza ai valori impliciti di default;

- Per ogni successiva chiamata ad un costruttore, cominciando dal costruttore individuato per primo, viene eseguita la sequenza di passi:

    1. Assegna i parametri del costruttore;
    2. Se è presente come prima riga, una chiamata esplicita a `this(...)`, richiama ricorsivamente il nuovo costruttore e poi passa al punto 5;
    3. Richiama l'implicito o esplicito `super(...)-costruttore`, eccetto per Object (non presenta superclassi);
    4. Esegue ogni inizializzazione esplicita delle variabili di istanza;
    5. Esegue il corpo del costruttore corrente.

## 12.13 La classe `Object`

È la radice di tutte le cassi. Una dichiarazione di classe senza la clausola `extends`, implicitamente usa *extends Object*.

Questo permette di sovrapporre numerosi metodi della classe *Object*.

### 12.13.1 Il metodo `equals`

- L'operatore `==` determina se 2 riferimenti sono identici (cioè se si riferiscono allo stesso, unico, oggetto);
- Il metodo `equals` determina se le due variabili si riferiscono ad oggetti (anche distinti) appartenenti alla stessa classe di equivalenza, rispetto ad una particolare relazione di equivalenza definita dall'utente;
- La realizzazione di equals nella classe *Object* fa uso di `==` $\rightarrow$ quindi le classi di equivalenza di `==` e di `equals` coincidono in *Object* (relazione di identità);
- Lo scopo del metodo è quello di essere sovrapposto nelle classi sviluppate dagli utenti, in modo da realizzare nuove relazioni di equivalenza;
- Alcune classi di sistema lo fanno già (String);
- Viene raccomandato di sovrappore, insieme con *equals*, anche il metodo *hashCode*.

### 12.13.2 Il metodo `toString`

Restituisce una rappresentazione `String` di un qualunque oggetto. Viene usato durante le conversioni automatiche a stringhe.

Occorer sovrappore questo metodo quando si vuole fornire una rappresentazione di un oggetto in forma (umanamente) leggibile.

I tipi primitivi sono rappresentati in forma di *String* usando il metodo statico `toString` della corrispondente classe *wrapper*

### 12.14 Le classi `wrapper`

I tipi primitivi non sono oggetti → Se si vogliono manipolare come oggetti, è possibile avvolgre un singolo valore con un opportuno oggetto, il cosiddetto *wrapper*:

| Tipo primitivo | Classe wrapper |
|---|---|
| boolean | Boolean |
| byte | Byte |
| char | Character |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

#### 12.14.1 Uso

Si può costruire un oggetto wrapper, passando il relativo valore al costruttore appropriato.

- Il valore da avvolgere può essere passato anche in una rappresentazione sotto forma di `String`;
- Il valore avvolto può essere estratto il metodo opportuno `...Value()`;
- Le classi wrapper sono utili quando si vogliono convertire i tipi primitivi.

### 12.15 Molteplicità

Si sfruttano gli arraylist.

## 13 Lezione del 26-04

### 13.1 Polimorfismo per inclusione (basato su upcast e downcast)

È uno strumento flessibile per realizzare strutture dati polimorfe.

Tuttavia il downcast necessario per estrarre elementi dalla struttura dati è verificabile solo a tempo di esecuzione (può generare errori a runtime che il compilatore non può identificare in anticipo).

### 13.2 Modificatori di accesso

Sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi. Un gruppo di modificatori possono essere usati, in combinazione con i precedenti, per qualificare quella caratteristica.

### 13.2.1 Generalità

Una caratteristica di una classe può essere:

- La classe stessa;
- Un attributo (variabile) della classe;
- Un metodo o un costruttore della classe.

Le sole variabili che possono essere controllate con i modificatori di accesso sono gli attributi (variabili di head) e non le variabili dei metodi (variabili di stack): una variabile di un metodo può essere vista solo dal metodo in cui si trova.

I metodi di accesso (mutualmente esclusivi) sono:

- public;
- protected;
- private.

Il solo modificatore di accesso permesso per una classe top-level è public: non esistono classi top-level protected o private.

Se non è presente il modificare, si intende "accesso consentito dallo stesso pacchetto in cui è situato quella caratteristica".

`public`

Modificatore più generoso. Accesso consentito a tutte le altre classi.

L'accesso ad un attributo o un metodo public è subordinato all'accesso alla classe che lo contiene

`private`

È il metodo meno generoso. Può essere usato solo per attributi o metodi, non per classi top-level

Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

```
class Complesso {
    private double reale , immag ;
```

```
}
class SubComplesso extends Complesso {

    SubComplesso(double r, double i){
real = r; // Illegale
    }
}
```

La classe SubComplesso eredita gli attributi della superclasse, ma quegli attributi possono essere usati solo dal codice della classe Complesso.

`Default`

Non è una parola riservata → si intende accesso di default quando non è presente nessun modificatore di accesso. Accesso consentito a tutte le classi nello stesso package.

`protected`

Accesso consentito a tutte le classi nello stesso package e a tutte le sottoclassi in pacchetti diversi, ma solo per ereditarietà: una sottoclasse in un altro pacchetto può accedere solo attraverso un riferimento ad un oggetto del proprio tipo (anche all'oggetto corrente, mediante `this`, o, esplicitamente, alla sua parte ereditata mediante `super`) o di un sottotipo.

### 13.2.2 Altri modificatori

`final`

Si applica a classi, metodi e variabili (non a costruttori).

Il significato varia da contesto a contesto, ma l'essenza è:

una caratteristica `final` non può essere modificata

Una classe final non può essere estesa, cioè non può avere sottoclassi.

Una variabile final è praticamente una costante, cioè può solo essere inizializzata.

**NB**: Un riferimento final ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento si.

Un metodo final non può essere sovrapposto in una sottoclasse.

Non ha senso dire `final` ad un costruttore, perchè esso non è mai eridatato dalle sottoclassi, quindi mai sovrapponibile

### `abstract`

Si applica solo a classi e a metodi. Un metodo abstract non possiede corpo (";" invece di "{...}").

Una classe può essere marcata abstract. In questo caso il compilatore suppone chessa contenga (anche per ereditarietà) metodi `abstract`: essa non potrà essere istanziata, anche se non contiene affatto metodi abstract.

Una classe deve essere marcata abstract se:

- Essa contiene almeno un metodo `abstract`, oppure;
- Essa eredita almeno un metodo `abstract`, per il quale non fornisce una realizzazione, oppure;
- Essa dichiara di implementare una interfaccia, ma non fornisce una realizzazione di tutti i metodi di quell'interfaccia.

In un certo senso `abstract` è opposto a `final` →

- Una classe final, per esempio non può essere specializzata;
- Una classe abstract, esiste solo per essere specializzata.

### `static`

Si applica ad attributi, metodi ed anche a blocchi di codice che non fanno parte di metodi. In generale, una caratteristica `static` appartiene alla classe, non alle singole istanze → essa è unica, indipendentemente dal numero (anche 0) di istanze di quella classe.

#### Attributi static

L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe)

```
class Ecstatic{
    static int x = 0;
```

```
    Ecstatic () {
x++;
    }
}
```

L'accesso ad un attributo `static` di una classe può avvenire (con la dot-notation):

- O partendo da un riferimento ad una istanza di quella classe;
- O partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);
Ecstatic e = new Ecstatic();
e.x = 100;
Ecstatic.x = 100;
```

### Metodi static

Esistono nel momento in cui la classe viene caricata in memoria (anche senza istanze).

Non possono accedere a membri non static della stessa classe (perchè potrebbero non esistere). Un esempio è il metodo main

```
class Foo {
    static int i = 48;
    int j = 1
public static void main (String args[]){
i += 100;
// j *= 5 ; Per fortuna è commentata
    }
}
```

- Non possono essere sovrapposti da metodi `non-static`;
- Non possono sovrapporre metodi `non-static`.

### Blocchi static

È lecito che una classe contenga blocchi di codice marcati `static`. Tali blocchi sono eseguiti una sola volta, nell'ordine in cui compaiono, quando la classe viene caricata in memoria ed, ovviamente, possono accedere (come i metodi `static`) solo a caratteristiche `static`

```
public class EsempioStatic{
    static double d = 1.23;
    static {
System.out.println("Codice static: d= " + d++);
    }

    public static void main (String args[]){
System.out.println("Main: d= " + d++);

    }

    static {
System.out.println("Codice static: d= " + d++);
    }
}
```

### native

Si applica solo a metodi. Come per `abstract`, `native` indica che il corpo di un metodo deve essere trovato altrove, in questo caso all'esterno della JVM, in una libreria di codice dipendente dalla architettura fisica.

### transient

Si applica solo agli attributi. Indica che quell'attributo contiene informazioni sensibili e che, nel caso in cui lo stato interno dell'oggetto debba essere slavato (nel gergo "serializzato"), il valore di quell'attributo non dovrà essere considerato.

### synchronized

Si applica solo a metodi o a blocchi anonimi.

Controlla l'accesso al codice in programmi multi-thread.

### volatile

Si applica solo agli attributi. Avverte il compilatore che quell'attributo può essere modificato in modo asincrono, in architettura multiprocessore.

### 13.2.3 Sommario

| Modificatore | Classe | Attributo | Metodo | Costruttore | Blocco |
|---|---|---|---|---|---|
| public | ✓ | ✓ | ✓ | ✓ | |
| protected | | ✓ | ✓ | ✓ | |
| "default" | ✓ | ✓ | ✓ | ✓ | ✓ |
| private | | ✓ | ✓ | ✓ | |
| final | ✓ | ✓ | ✓ | | |
| abstract | ✓ | | ✓ | | |
| static | | ✓ | ✓ | | ✓ |
| native | | | ✓ | | |
| transient | | ✓ | | | |
| volatile | | ✓ | | | |
| synchronized | | | ✓ | | ✓ |

### 13.2.4 Singoletto

Uno dei problemi ricorrenti è il seguente → si vuole che una classe abbia una sola istanza. Se si scrivesse la classe nel classico modo, gli utilizzatoripotrebbero creare istanze a volontà.

Lo scopo della forma progettuale *Singleton* è quello di assicurare il progettista della claase che esisterà sempre una ed una sola instanza di quella classe

# 14 Lezione del 30-04

## 14.1 Classi astratte

Java permette ad una superclasse di dichiarare un metodo, del quale non fornirà la realizzazione. Questa invece verrà fornita dalle sottoclassi.

Tali metodi sono *astratti* e una classe con uno o più metodi astratti è una *classe astratta*.

Java proibisce la costruzione di oggetti di una classe astratta, tuttavia classi astratte possono avere attributi, metodi concreti e costruttori. È buona norma rendere questi costruttori *protected* piuttosto che *public*.

Un'altra possibile soluzione consiste nel porre in una classe astratta un metodo concreto, che utilizzi metodi astratti nella stessa classe. Poichè tale tecnica è assai ricorrente, essa viene comunemente indicata come *Metodo Sagoma* (*Template Method Design Pattern*).

## 14.2 Interfacce

### 14.2.1 Generalità

Una *interfaccia* è un contratto tra il codice cliente e la classe che *implementa* quell'interfaccia. In Java è una formalizzazione di un tale contratto in cui tutti i metodi non contengono realizzazioni.

Molte classi, anche non correlate, possono implementare la stessa interfaccia.

Una classe può implementare molte interfacce, anche non correlate.

Una interfaccia può estendere altre interfacce.

### 14.2.2 Sintassi

```
<class_declaration> ::=
  <modifier> class <name> [extends <superclass>]
    [implements <interface> [, <interface>]*] {
      <class_body>
}

<interface_declaration> ::=
  <modifier> interface <name>
    [extends <interface> [, <interfac>]*]{
<interface_body>
}
```

**NB** → una interfaccia può anche dichiarare costanti:

```
public static final int COSTANTE = 8;
```

**Attenzione** → le variabili dichiarate nelle interfacce sono implicitamente *public static final*. Pertanto la dichiarazione precedente poteva anche essere scritta:

```
int COSTANTE = 7;
```

### 14.2.3 Vantaggi delle interfacce

Le interfacce sono spesso considerate una alternativa all'eredità multipla, anche se esse forniscono diverse funzionalità. Esse sono utili:

- Per dichiarare metodi che una o più classi ci si aspetta dovrà implementare;

- Per rivelare solo una ... interfaccia, senza rivelare il corpo vero di una classe (per esempio quando si distribuisce una classe ad altri sviluppatori di codice);
- Per catturare similarità tra classi non correlate, sena forzare una relazione tra classi;
- Per simulare l'ereditarietà multipla, dichiarando una classe che implementi varie interfacce.

## 14.3 Casting di riferimenti

I riferimenti, come i primitivi, partecipano alla conversione automatica per asegnamento (e per passaggio di parametri) e al casting.

Non c'è promozione aritmetica di riferimenti, poichè i riferimenti non possono essere operandi aritmetici.

Nella conversione e nel casting di rifermineti vi sono maggiori combinazioni tra vecchi e nuovi tipi e maggiori combinazioni significano un numero maggiore di regole.

La conversione (automatica) di riferimenti avviene nella fase di compilazione, poichè il compilatore ha tutte le informazioni se la conversione è legale.

Essa può essere forzata con un casting, come per i primitivi. In questo caso, può accadere che sebbene la conversione sia autorizzata come legale dal compilatore, il tipo effettivo dell'oggetto riferito non sia compatibile in esecuzione (in una assegnazione entrano in gioco 3 tipi → quello dei riferimenti e quello proprio dell'oggetto).

### 14.3.1 Conversioni automatiche (1)

Nella conversione automatica di riferimenti (per ampliamento, in una assegnazione o in un passaggio di parametri), vengono nascoste alcune caratteristiche dell'oggetto riferito.

Qui il tipo di nascita dell'oggetto non interessa; i riferimenti possono essere:

- A una classe;
- Ad una interfaccia;
- Ad un array.

La conversione può avvemire in:

```
Oldtype = new Oldtype();
Newtype y = x;
```

| | Oldtype è una classe | Oldtype è un interfaccia | Oldtype è un array |
|---|---|---|---|
| Newtype è una classe | Oldtype deve essere una sottoclasse di Newtype | Newtype deve essere Object | NewType deve essere Object |
| Newtype è una interfaccia | Oldtype deve implementare l'interfaccia Newtype | Oldtype deve essere una sottointerfaccia di Newtype | Newtype deve essere Cloneable o Serializable |
| Newtype è un array | Errore di compilazione | Errore di compilazione | I tipi delle componenti dei 2 array devono essere riferimenti auto-convertibili |

### 14.3.2 Conversioni automatiche (2)

Regola breve:

- Un tipo interfaccia può essere convertito solo ad un tipo interfaccia o ad `Object`. Se il nuovo tipo è un interfaccia essa deve essere una superinterfaccia i quella del vecchio tipo;
- Un tipo classe può essere una superclasse del vecchio tipo. Se convertito ad un tipo interfaccia, la vecchia classe deve implementare l'interfaccia;
- Un array può essere convertito alla classe `Object`, all'interfaccia Cloneable o all'interfaccia `Serializable`, oppure ad un array. Solo una array di riferimenti (non di primitivi) può essere convertito ad un array, e il vecchio tipo degli elementi deve essere convertibili al nuovo tipo degli elementi.

### 14.3.3 Casting

Una volta appurato quali sono le conversioni che il compilatore accetta di fare da sè (quella per assegnazione, o per passaggio di parametri, di ampliamento), possiamo pasare a studiare le conversioni che il programma chiede esplicitamente (casting)

Un *cast di ampliamento*, la cui conversione sarebbe avvenuta anche senza di esso, viene autorizzato e non causa problemi nè di compilazione, nè in esecuzione.

DEtto rozzamente, un casting di riferimenti verso l'alto, nella gerarchia di ereditarietà, è sempre, sia implicitamente sia esplicitamente, legale.

Purtroppo per i riferimenti e a differenza dei primitivi, nel casting verso il basso, il compilatore non può aiutare complementamente e può accadere che, sebbene la conversione sia autorizzata come legale, essa fallisca in esecuzione.

Infatti, nel casting esplicito, entrano in gioco 3 tipi:

- Quello dei 2 riferimenti (a sinistra e a destra dell'assegnazione);
- Quello vero (l'identità, l'imprinting di nascita) dell'oggetto.

### 14.3.4 Casting esplicito

Il casting esplicito avviene quando

```
Newtype nt;
Oldtype ot;
nt = (Newtype) ot;
```

Regole per il compilatore

| | Oldtype è una classe non final | Oldtype è una classe final | Oldtype è un interfaccia | Oldtype è un array |
|---|---|---|---|---|
| Newtype è una classe non-final | Oldtype deve estendere Newtype o viceversa | Oldtype deve estendere Newtype | Sempre OK | Newtype deve essere Object |
| Newtype è una classe final | Newtype deve estendere Oldtype | Oldtype e Newtype devono coincidere | Newtype deve implementare l'interfaccia o implementare Serializable | Errore di compilazione |
| Newtype è una interfaccia | Sempre OK | Oldtype deve implementare l'interfaccia Newtype | Sempre OK | Errore di compilazione |
| Newtype è un array | Oldtype deve essere Object | Errore di compilazione | Errore di compilazione | I tipi delle componenti dei 2 array devono essere riferimenti convertibili per cast |

### 14.3.5 Casting - Regole brevi

Regola breve (per il compilatore, nei casi più comuni):

- Quando sia `Oldtype`, sia `Newtype` sono classi, una classe deve essere sottoclasse dell'altra;

- Quando sia `Oldtype`, sia `Newtype` sono array, entrambi devono contenere riferimenti (non primitivi), e deve essere legale eseguire un cast tra gli elementi di `Oldtype` e quelli di `Newtype`;
- È sempre legale il cast tra una interfaccia e una classe non-final.

#### 14.3.6 Casting - Regole ulteriori

Infine le ulteriori regole a cui deve obbedire un cast durante l'esecuzione (posto che sia sopravvisuto alla compilazione):

- Se `Newtype` è una classe, la classe originaria `Objtype` dell'oggetto deve essere Newtype oppure una sua sottoclasse.
- Se `Newtype` è un interfaccia, la classe originaria `Objtype` dell'oggetto deve implementare `Newtype`.

# 15 Lezione del 04-05

## 15.1 String

Le stringhe sono oggetti che è possibile creare nei seguenti modi:

```
String s = new String();
s = "abcdef";

//

String s = new String("abcdef");

//

String s = "abcdef";
```

La cosa da ricordare è che le stringhe sono oggetti immutabili → Significa che, una volta assegnato all'oggetto un valore, esso è fissato per sempre.

**NB:** Immutabili sono solo gli oggetti String, non i loro riferimenti (questi possono variare).

### 15.1.1 String constant pool

Per motivi di efficienza, poichè nelle applicazioni i litterali String occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la `String constant pool`.

Quando il compilatore incontra un litterale String, esso controlla che non sia già presente nel pool.

Se presente allora il riferimento al nuovo litterale è diretto alla stringa esistente (la stringa esistente ha semplicemente un ulteriore riferimento), altrimenti lo aggiunge al pool.

Questo rende chiaro il concetto di stringhe come oggetto immutabile. Se ci sono molti riferimenti, in punti diversi del codice, allo stesso litterale, sarebbe deleterio permettere la modifica del litterale usando uno solo di tali riferimenti.

### 15.2 StringBuffer

Se si deve fare uso intensivo di manipolazione di stringhe, allora è opportuno usare oggetti di tipo **StringBuffer**: essi sono un po' come gli oggetti String, ma non sono immutabili.

*es.*

```
StringBuffer s = new StringBuffer("Marcello");
s.append(" Sette");
System.out.println(s); // Stampa Marcello Sette
```

**Attenzione**:

```
StringBuffer s = "abc";
// Illegale: String e StringBuffer non sono convertibili

StringBuffer s = (StringBuffer) "abc";
// Illegale: neanche con cast!
```

**Attenzione**: Mentre la classe String sovrappone il metodo `equals` in modo da controllare l'uguaglianza del contenuto dei 2 oggetti (quello corrente e quello ricevuto come parametro), la classe StringBuffer non lo sovrappone ed usa quello ereditato da Object che funziona essenzialmente come l'operatore `==` (cioè compara i riferimenti).

Le classi `String` e `StringBuffer` sono `final`. Esse non possono essere specializzate in modo da sovrapporre i loro metodi: L'invocazione virtuale di metodi sarebbe un grave problema di sicurezza.

## 16 Lezione del 11-05

### 16.1 Le eccezioni (Gestione degli errori)

#### 16.1.1 Meccanismo

Le eccezioni denotano eventi eccezionali la cui occorrenza altera il flusso normale delle istruzioni, ad *es*:

- Risorse HW indisponibili;

- HW malfunzionante;
- Bachi nel SW.

Quando capita tale evento, si dice che viene "lanciata una eccezione".

### Try e catch

- Il codice che potrebbe lanciare una eccezione è inglobato in un blocco marcato *try*;
- Il codice che assume la responsabilità di fare qualcosa in conseguenza del lancio di una eccezione si chiama *manipolatore* (exception handler) e va inglobato in una clausola *catch*.

*es*

```
try {
    // Qui va inserito il codice "rischioso"
} catch (Eccezione1 e){
    // Qui il codice che manipola una Eccezione1
} catch (Eccezione2 e){
    // Qui il codice che manipola una Eccezione1
}
// Qui va scritto il codice non rischioso
```

### Finally

Un blocco opzionale *finally* verrà (quasi) sempre eseguito, anche dopo il lancio ed la eventuale manipolazione (eventuale) dell'eccezione.

```
try {
    // Qui va inserito il codice "rischioso"
} catch (Eccezione1 e){
    // Qui il codice che manipola una Eccezione1
} catch (Eccezione2 e){
    // Qui il codice che manipola una Eccezione1
}finally {
    // Codice da eseguire in ogni caso
}
// Qui va scritto il codice non rischioso
```

Il blocco *finally* viene eseguito perfino dopo una eventuale istruzione *return* present nei blocchi *try* o *catch*.

Il blocco *finally* viene eseguito sempre, eccezion fatta nel caso di un crash totale del sistema oppure tramite una invocazione di *System.exit(int status)*.

### 16.1.2 Vincoli

Le clausole *catch* e il blocco *finally* sono opzionali. Dopo un blocco *try* deve esistere almeno una clausola *catch* oppure un blocco *finally*. Un blocco *try* solitario causa un errore di compilazione.

Se esistono una o più clausole *catch* esse devono seguire immediatamente il blocco try.

Se esiste il blocco *finally* esso deve seguire l'ultima clausola *catch*.

Non è ammessa nessuna istruzione tra il blocco *try*, le clausole *catch* ed il blocco *finally*

È significativo l'ordine in cui si succedono tra loro le clausole *catch*.

### 16.1.3 Propagazione

Una eccezione non catturata semplicemente si immerge nel record di attivazione che la ha generata, "riemengendo" nel successivo record di attivazione.

Qui potrebbe essere catturata da una ulteriore clausola *catch*, oppure altrimenti ricadere nel record successivo, e così via. Finchè, eventualmente, l'eccezione raggiunge il record di attivazione del main, dal quale, se non catturata, "esplode", producendo (forse) una pittoresca descrizione del proprio percorso (stack trace).

### 16.1.4 Definizione

In Java tutto ciò che non è primitivo è un oggetto. Ogni eccezione infatti è una istanza di una sottoclasse della classe *Exception*.

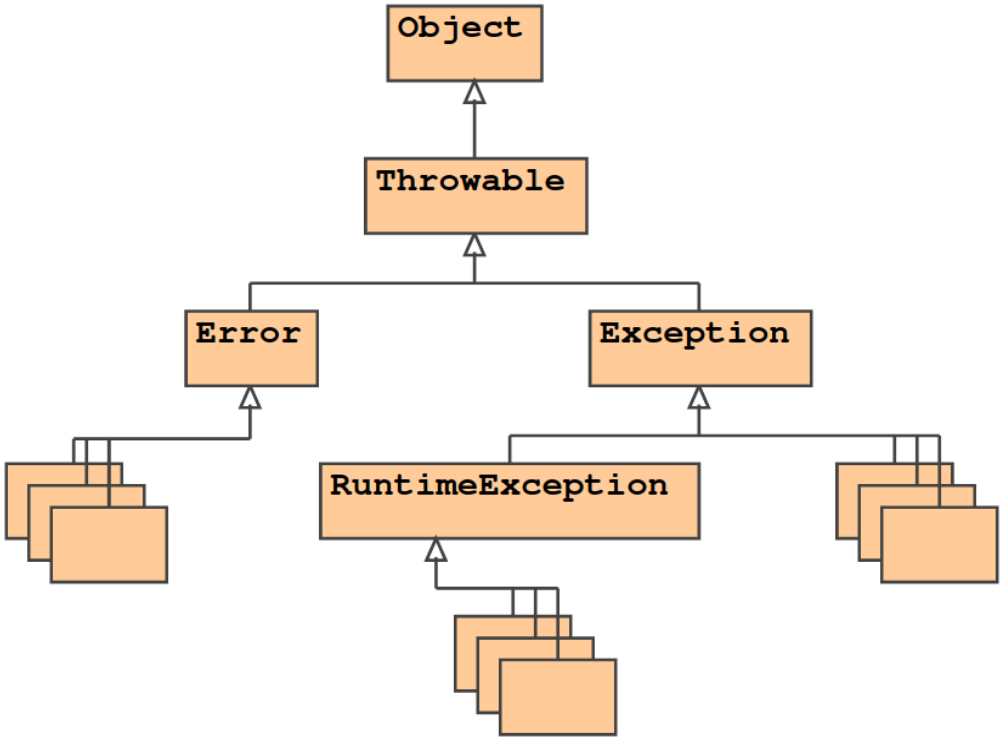
Una eccezione viene lanciata usando la parola riservata *throw*:

```
throw new Exception();
```

Tutto ciò che segue, nello stesso blocco di istruzioni, il lancio dell'eccezione non verrà eseguito (tranne l'eventuale blocco *finally*).

### 16.1.5 Gerarchia

La gerarchia degli oggetti lanciabili è la seguente:




La classe *Throwable* rappresenta tutti gli oggetti che possono essere lanciati. Essa contiene il metodo `printStackTrace`.

La classe *Error* e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Ad esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa risulta non disponibile.

In genere, una applicazione non deve essere capace di riprendersi da una situazione del genere. Pertanto un programma non è obbligato a gestire gli oggetti *Error*: esso compila senza problemi.

La classe *RuntimeException* rappresenta pure eventi eccezionali, ma dovuti al programma (errore di programmazione, bachi). Il programmatore, che si accorge di un baco dovuto ad un suo errore deve correggerlo, e non gestirlo come una eccezione.

### 16.1.6 Eccezioni catturate

Una clausola *catch* cattura ogni oggetto-eccezione il cui tipo può essere ricondotto mediante conversione automatica al tipo specificato nella clausola.

*es.* La classe `IndexOutOfBoundException` ha due sottoclassi (`ArrayIndexOutOfBoundException` e `StringIndexOutOfBou
→ si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:

```
try{
    // Codice che potrebbe lanciare una eccezione
    //IndexOutOfBoundException
    //ArrayIndexOutOfBoundException
    //StringIndexOutOfBoundException
}catch (IndexOutOfBoundException e){
    e.printStackTrace()
}
```

Cose da evitare:

- Scrivere una unica clausola catch-all;
- Scrivere catch che si restringono come ampiezza (vanno messe prima le clausole con dei catch più piccoli).

Così come la dichiarazione del metodo deve specificare il numero e il tipo di parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclasi di RunTimeException)

La parola chiave *throws* viene usata per elencare le eccezioni che possono fuoriuscire da un metodo.

Il fatto che un metodo dichiara l'eccezione non significa che esso la lancerà sempre, ma avverte l'utilizzatore che esso potrebbe lanciarla.

## 16.2 Handle or Declare

Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *hangle or declare*):

1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`;
2. Propagandare l'eccezione dichiarandola nell'intestazione del metodo.

Una *eccezione* a questa regola → le RunTimeException sono esenti dall'obbligo di dichiarazione. Esse sono *unchecked* dal compilatare (le restanti no).

**NB**: l'*or* della regola non è esclusivo (si può decidere di fare entrambe le cose).

## 16.3 Nuove eccezioni

È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

```
class MiaEccezione extends Exception {}
```

Oppure estendendo una qualunque sottoclasse di Exception.

Da questo momento in poi si può lanciare un oggetto del tipo (checked) MiaEccezione.

### 16.4 Overridingi - Regole

- I 2 metodi devono avere identica signature;
- I 2 metodi devono avere identico tipo di ritorno;
- Non si può marcare *static* uno solo dei metodi;
- Il metodo nella superclasse non può essere marcato *final*;
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse;
- Il metodo nella sottoclasse può dichiarare di lanciare un tipo di eccezione *checked*, ma tale tipo non deve essere un nuovo tipo o un tipo più esteso rispetto a quelli dichiarati dal metodo della superclasse.

Cioè, le eventuali eccezioni *checked* dichiarate dal meotodo nella sottoclasse, devono essere di tipi posti al di sotto nella gerarchia delle eccezioni dichiarate dal metodo nella superclasse.

## 17 Lezione del 18-05

### 17.1 Gestione degli eventi

Piuttosto che controllare ciclicamente e continuamente lo stato di un oggetto (un pulsante ad esempio) si scegli un approccio on demand.

È molto più efficiente infatti lasciare che sia l'oggetto a comunicare il proprio cambiamento di stato:

Il nostro codice diventa un *ascoltatore* e il pulsante è la *sorgente*.

### 17.2 Classi interne (classi nested)

Danno al programma maggiore chiarezza e lo rendono più coinciso. Offrono anche una soluzione al problema relativo all'ereditarietà multipla.

Sono come le altre classi, ma sono dichiarate all'interno di altre classi. Possono essere poste in qualunque blocco, incluso i blocchi dei metodi.

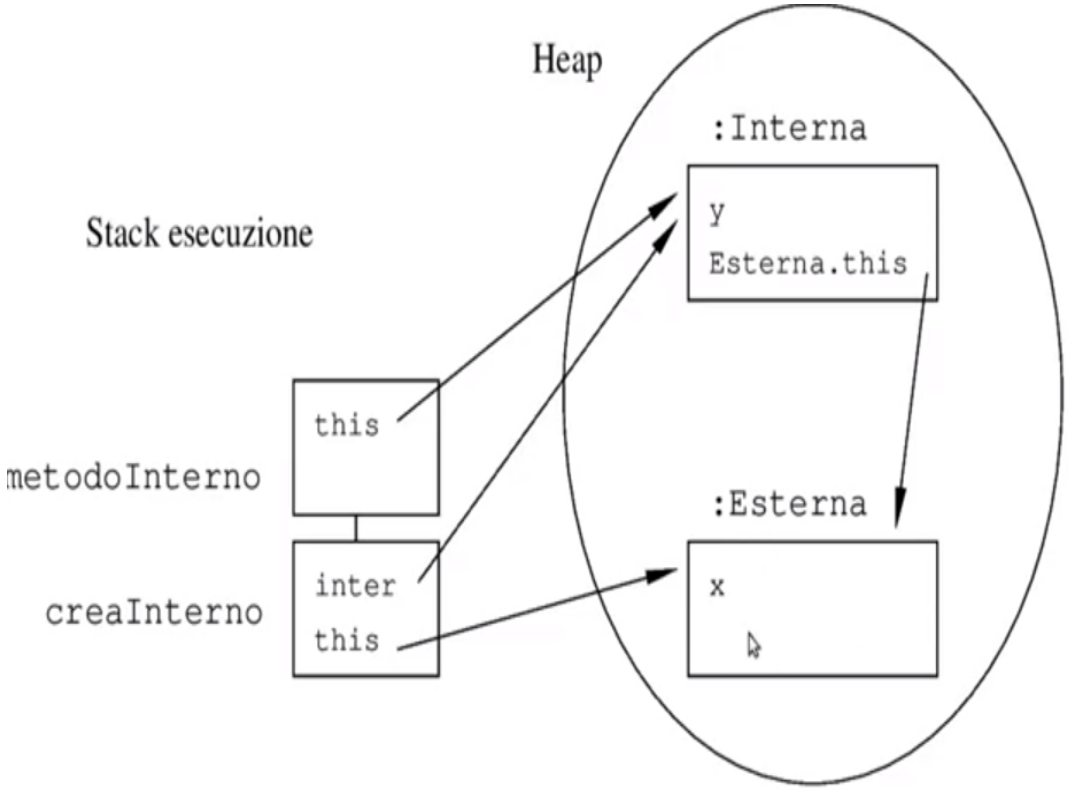
Il vantaggio delle classi interne è la possibilità di poter dare a più classi interne su classi diverse (file diversi) lo stesso nome (si hanno infatti namespace diversi)

### 17.2.1 Creazione di una istanza di una classe Interna

Quando si deve costriure una istanza di una classe Interna, in genere, deve esistere già una istanza della classe esterna che agisca da contesto.

L'oggetto interno può accedere a tutte le componenti dell'oggetto esterno, indipendentemente dalla loro visibilità, attraverso un riferimento implicito.

L'ogetto esterno, avendo costruito quello intenro, ne possiede un riferimento. L'oggetto interno, però, può accedere all'oggetto esterno attraverso un riferimento implicito (tale riferimento può essere anche esplicitato)




### 17.2.2 Modificatori

Classi membro possono essere opzionalmente marcate con qualunque modificatore di accesso (a differenza delle classi top-leve, che possono essere opzionalmente marccate solo *public*).

Il significato è identico a quello degli altri membri della classe (attributi o metodi).

Ad esempio, una classe membro *private* può essere vista solo all'interno della classe includente → non si può nominarla al di fuori della classe includente.

Classi membro *final* non possono essere specializzate.

Classi membro *abstract* non possono essere istanziate.

Classi membro *static*:

- Esistono a prescindere dell'oggetto includente;
- Non hanno il riferimento implicito all'oggetto includente.

## 17.3 Classe definita all'interno di un metodo

Come tutte le altre variabili del metodo, sonon locali in quel metodo e non possono essere marcate con nessun modificatore tranne che con *abstract* o *final*.

Esse non sono accessibili in alcun modo all'esterno del metodo. Presentano accesso limitato alle altre variabili locali del metodo

### 17.3.1 Accesso a variabili locali

Ogni variabile locale del metodo includente non può essere utilizzata dalla classe interna, a meno che quella variabile non sia marcata *final*.

In realtà un oggetto (sullo heap) non dovrebbe accedere affatto alle variabili dei metodi (di stack) poichè gli oggetto sullo heap possono sopravvivere ai record sugli stack.

**NB**: La variabile *final* è una costante nel metodo. L'oggetto che deve accedervi ne possiede semplicemente una copia. In tal modo, alla terminazione del metodo, la costante sarà ancora utilizzabile, anche se non esiste più il record di attivazione che la contiene.

## 17.4 Classi anonime

Possono essre usate per estendere un'altra classe oppure, in alternativa, per implementare una singola interfaccia.

La sintassi non concede di fare entrambe le cose, nè di implementare più di un interfaccia → se si dichiara una classe che implementa una singola interfaccia, allora la classe è sottoclasse diretta di `java.lang.Object`.

Poichè non si conosce il nome di una tale classe, non si può usare `new` nel modo solito per crearne una istanza.

Infatti, la definizione, la costruzione e il primo uso (spesso in una assegnazione) avvengono nello stesso enunciato.

L'utilità è nella possibilità di sovrapporre qualche metodo della superclasse (o di implementare metodi di una interfaccia) senza la necessità di scrivere una vera classe che lo faccia.

```
class A {
    public void f(){
```

```
System.out.println("A");
    }
}
class B {
  A a = new A(){
public void f(){
    System.out.println("B");
}
    };
}
```

In questo caso la variabile `a` si riferisce non ad una istanza di A, ma *ad una istanza di una sottoclasse anonima di A*.

# 18 Lezione del 21-05

## 18.1 Paradigma funzionale

Paradigma in cui i programmi sono funzioni e la valutazione di questi programmi si riduce al calcolo di espressioni.

Programmare in sitle funzionale puro significa usare solo espressioni e funzioni, eventualmente ricorsive.

Non vi sono assegnamenti (non c'è una memoria che cambia):

- Perchè gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare).

Quindi senza assegnamenti non possono esistere cicli `while/for`.

### 18.1.1 Conseguenze alla programmazione

- Ricorsione al posto dei cicli;
- Modifiche all'ambiente anzichè alla memoria;
- Creazione identificatori con stesso nome mediante chiamate ricorsive;
- Creazione di nuovi identificatori con lo stesso nome che mascherano la versione precedente (come nello scoping statico).

## 18.2 ML

Linguaggio che supporta:

- Interprete interattivo;
- Implementazione mista (compilazione su codice intermedio e successiva interpretazione);
- Compilazione su codice oggetto *standalone*.

È un linguaggio fortemente e staticamente tipato:

- Il controllo dei tipi avviene interamente a tempo di compilazione.

Non richede di dichiarare il tipo di identificatore → spesso lo capisce da solo (*type inference*).

- Usa sia *structural equivalence* che *name equivalence*;
- Permette di definire tipi ricorsivi (liste, alberi, ...);
- Supporta *polimorfismo parametrico* (come i *template*);
- Ha un garbage collector;
- Supporta *encapsulation* (tipi di dato astratto) ma non è un linguaggio OO
  - Mancano la gerarchia di tipi e, di conseguenza, l'ereditarietà

### 18.2.1 Uso

Puo essere utilizzato in 2 modi

- Interagendo con l'interprete:
  - Inserendo definizioni ed espressioni una per una;
  - L'interprete risponde ad ogni passo;
  - Si possono caricare programmi da file digitando nella shell dell'interprete il comando `use "nome del file";` (questo rende utilizzabile le dichiarazioni contenute nel file)
- Compilando un programma in codice oggetto direttamente eseguibile
  - Ad *es* attraverso il compilatore `mlton` per standard ML → `mlton "nome del file.sml"` (che produce un file eseguibile con lo stesso nome senza estensione).

### 18.2.2 Tipi primitivi

- Int → 1, 2, `~2` (nel caso fosse negativo), `0xff, ~0x32`
  - Gli operatori → `+, -, *, div, mod, <,...`
- Word (unsigned integers) → `0w44, 0w14, 0wxff`;

- Real;
- String → "abc", "123"
  - Operazioni → `^` (concatenazione), `size, <, ...`;
- Char → `#"a", #"\n", #"\163"`
  - Operazioni → `ord, chr, <, ...` (ord → da carattere ne restituisce il codice);
- Bool
  - Operazioni → `not, andalso, orelse,...`;

**NB**: Nessuna conversione automatica tra i tipi nmerici → Va usato:

- real:int → real;
- Basis library (presenti per ogni tipo primitivo con funzioni per conversione, parsing e altre utilità).

Real non supporta l'uguaglianza → Usare Real.==;

Questo perchè lo standard IEEE prevede valori che risultano da operazioni non definite, denominati **NaN** (not a numer) → questo tipo non è confrontabile con nessun altro numero, nemmeno con sè stessi.

### 18.2.3 Funzioni

Ci sono diversi modi di definire e chiamare funzioni in ML. Ad *es*:

```
fun fatt x = if x=0 then 1 else x*fatt(x-1);

fatt(3); # chiamata alla funzione
fatt 3; # chiamata alla funzione (in questo caso le parentesi sono opzionali)

fatt; # val it = fn : int -> int
```

Con fun si dichiara la funzione:

- fun aggiunge all'ambiente l'identificatore *fatt*;
- Lo associa alla funzione da interi a interi.

Si può vedere cosa è associato a *fatt* senza chiamare la funzione (chiamando fatt senza paramentri) → ci mostra solo il tipo (il valore è stato trasformato in bytecode).

Grammatica delle dichiarazioni

```
<declaration> ::=
    val <id name> = <expression> |
    fun <func name> <arguments>* = <expression>
```

(*val* è più generale di *fun*).

L'equivalente dei blocchi in ML è:

```
let
    <dichiarazioni>
in
    <espressione> (* le dichiarazioni valgono solo qui *)
end
```

(Lo scoping è statico)

### 18.2.4 Definizioni ausiliarie

**Locali ed altre definizioni**

```
local
    <dichiarazioni>
in
    <dichiarazione> (* le dichiarazioni valgono solo qui *)
end
```

Simile a *let* ma dopo in c'è una dichiarazione invece di un espressione da valutare.

### 18.2.5 Prodotti cartesiani

Si possono definire `n-uple` semplicemente mettengo i valori tra parentesi:

Il prodotto cartesiani viene indicato con `*`. Si estrae l'i-esimo elemento da una `n-upla` con `#i`.

### 18.2.6 Record

Insieme di espressioni *<nome>=<valore>*. Il valore associato al nome $N$ si estrae con `#N`.

### 18.2.7 Dichiarazioni di tipo

ML permette di definire nuovi tipi similmente ai typedef del C:

```
type coord = real * real;
```

In questo caso i tipe *coord* e *(real * real)* sono compatibili tra loro. Analogamente anche qualsiasi tipo definito *(real * real)* sarà compatibile con il tipo *coord*.

### 18.2.8 Datatype

Si possono definire costruttori per creare *data object*:

```
datatype color = red | green | blue;
```

```
val c = red; (* val c = red : color*)
```

Sembra molto simile alle enum del C (anche se non le prende in maniera seria in quanto con le enum usiamo in pratica interi).

I datatype di ML definiscono tipi nuovi.

Ogni tipo definito con datatype è incompatibile con tutti gli altri tipi (*name equivalence*).

### 18.2.9 Costruttori con Argomenti

*es.* Definire una lista concatenata

Se ne può dare una definizione ricorsiva → una lista di interi è:

- Una lista vuota (caso base);
- Un nodo che contiene un intero e una lista di interi

Servono quindi 2 costruttori → per la lista vuota e per i nodi:

```
datatype listaInt = vuota | nodo of int * listaInt;
```

```
val L = nodo(1, nodo(2,vuota));
```

Dichiaro in questo modo un *datatype listaInt = nodo of int * listaInt | vuota*.

Dichiaro un *val L = nodo (1, nodo (2,vuota)) : listaInt.*

*es* Definire un albero binario con nodi etichettati da interi

Definizione ricorsiva → un albero simile è:

- Un albero vuoto oppure;
- Un nodo che contiene un intero e 2 alberi dello stesso tipo

```
datatype albero = vuoto | of  nodoAlb of int * albero * albero

nodoAlb (1, nodoAlb (2, vuoto, vuoto), vuoto)
```

In questo caso creiamo un albero con radice 1, figlio sinistro 2 (che è una foglia), mentre il figlio destro è assente.

### 18.2.10 Patterns e matching

Per scandire una lista abbiamo inanzitutto bisogno di controllare se è vuota:

```
val L = nodo (1, nodo(2,vuota));

L = vuota; # falso in questo caso
```

Nel caso non sia vuota estraiamo il primo elemento con un *pattern matching*:

```
val nodo(p,_) = L; (* assegna a è il 1° elemento di L *)
val p = 1: int (* "_" è una wildcard *)
```

Per ottenere il presto della lista

```
val nodo(_, r) = L; (* assegna a r il resto * )
val r = nodo (2, vuota) : listaInt
```

Calcolo degli elementi di una lista

```
fun conta x = if x=vuota then 0
      else let val nodo(_, r) = x
```

```
      in conta (r) + 1 end;

val conta = fn : listaInt -> int (* risposta *)

conta L:

val it = 2: int (* risposta *)
```

È interessante notare che:

- Il compliatrore ha inferito il tipo della funzione
  - $x$ viene confrontato con "vuota", che è di tipo *listaInt* $\rightarrow$ anche $x$ è di tipo *listaInt* $\rightarrow$ l'input di *conta* è un *listaInt*;
  - Il *then* restituisce 0 che è un intero, quindi l'output di *conta* è un intero;
- Inoltre il compilatore controlla che anche il restod della funzione sia compatibile con questi tipi
  - $r$ corrisponde al 2° argomento del nodo, che è di tipo *listaInt* $\rightarrow$ è corretto passarlo a conta che restituisce un intero.

#### 18.2.11 Abbreviazioni per i pattern

Si possono estrarre tutti gli elementi di un costruttore in un colpo solo:

```
val nodo (p,r) = L;

val p = 1 : int ( * output *)
val r = nodo (2, vuota) : listaInt ( * output *)
```

Si può definire una funzione per casi:

```
fun conta vuota = 0
    | conta(nodo (_,r)) = conta(r) + 1;
```

Mettendo direttamente i pattern al postod dei parametri formali

Simil switch case:

```
case L of vuota => true
    | nodo(_, _)I => false;
```

# 19 Lezione del 25-05

## 19.1 Le liste

Le liste sono tra le strutture dati più utilizzate in programmazione funzionale (vanno a sostituire in qualche modo i vettori, che rappresentano un concetto essenzialmente iterativo).

Pertanto ML le fornisce built-in con i costruttori `nil` e `::`

```
nil 1 (* lista vuota *)

1 :: 2 :: 3 :: 4 :: nil (* lista vuota che contiene 1, 2, 3 *)

(* formato equivalennte basato su parentesi quadre *)
[]
[1,2,3]
```

### 19.1.1 Principali operatori sulle liste in ML

La funzione **lenght** restituisce la lunghezza di una stringa.

La funzione **null** restituisce true se la stringa è vuota.

La funzione **hd** e **tl** (head and tail) restituiscono il primo elemento e il resto della lista, rispettivamente:

```
val L = [1,2,3];

hd L; (* restituisce 1 *)

tl L; (* restituisce 2,3 *)
```

Altre funzioni si trovano nella struttura *List*, come:

- *List.nth* (L, $i$) → restituisce l'$i$-esimo elemento di L (partendo da 0).
- *List.last* (L) → restituisce l'ultimo elemento della lista $L$.

## 19.2 Curryng

Prende il nome dal matematico Haskell Curry che aveva sviluppato il tipo di analisi delle funzioni.

In ML ogni funzione ha un solo argomento:

```
fun f(x,y) = ... (* l'argomento è una singola coppia *)

fun f x y = ... (* l'argomento è x ! *)
```

Nel secondo caso, $f$ è una funzione che restituisce una funzione che prende y e calcola l'espressione dopo l'uguale.

*es.*

```
- fun f (x,y) = x+y
val f = fn : int * int -> int

- fun f' x y = x+y
val f' = fn : int -> int -> int
```

Il tipo $int \to int \to$ va inteso come $int \to (int \to int)$

La trasformazione da *n-ple* (come *f(x,y)*) a funzione che restituiscono funzioni (come *f' x y*) si chiama *curryng*

```
 - f(3,2); (* ottengo 5 *)

- f' 3 2; (* viene intepretato come (f' 3)(2) -> ottengo 5 *)
```

## 19.3 Funzioni di ordine superiore

La maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura. Cambia solo l'operazione che si applica ai nodi.

Quindi basta scrivere una sola volta la funzione che scandisce la struttura dati (che fa la funzione del ciclo) e passargli la funzione da applicare ai nodi.

Le funzioni che hanno altre funzioni come parametri sono dette di *ordine superiore*.

Le tipologie di `funzioni/ciclo` più comuni sono 3:

- Filter;
- Map;
- Reduce.

### 19.3.1 Funzione filter

Prende una funzione booleana $f$ e una lista $L$ e seleziona gli elementi di $L$ per cui $f$ è vera:

```
fun filter f [] = [] |
    filter f (x::y) = if f(x) then x::(filter f y)
else filter f y

(* esempio: seleziona gli elementi negativi da una lista *)
- let fun neg x = x < 0
    in filter neg [0, ~1, 3, ~2] end;
val it = [~1, ~2] : int list

(* esempio: seleziona gli elementi positivi da una lista *)
- let fun pos x = x > 0
    in filter pos [0, ~1, 3, ~2] end;
val it = [3] : int list
```

### 19.3.2 Funzione Map

La funzione map prende una funzione (qualunque) $f$ e una lista L e applica f a tutti gli elementi della lista:

```
fun map f [] = [] |
    map f (x::y) = f(x)::(map f y);

(* esempio: conversione in liste di reali *)

- map real [1,2,3];
val it = [1.0, 2.0, 3.0] : real list

(* esempio: conversione in liste di strighe *)

- map Int.toString [1,2,3];
val it = ["1", "2", "3"] : string list
```

### 19.3.3 Funzione Reduce

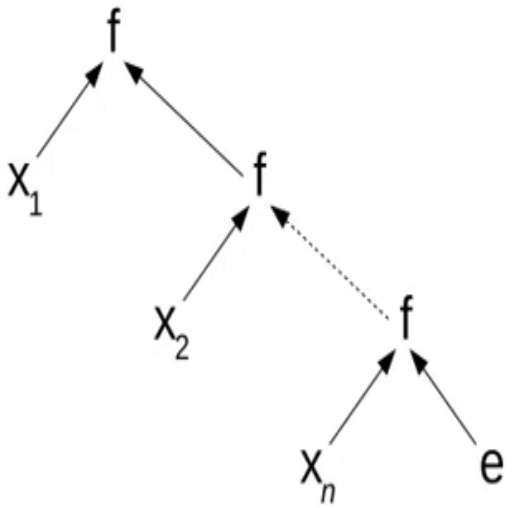
La funzione Reduce serve per calcolare aggregati di una lista:

- Min;
- Max;
- Somma;
- Prodotto;
- Media;

- ...

Prende in input una funzione a 2 argomenti *f*, un valore finale e una lista *L*, ed effettua il seguente calcolo:

$$\text{reduce } f\ e\ [x_1, x_2, \ldots, x_n] = f(x_1, f(x_2, \ldots f(x_n, e)\ldots))$$




Ad esempio se f è + ed *e* è 0, allora *reduce* calcola la somma degli elementi della lista.

```
fun reduce f e [] = e |
    reduce f e (x :: y) = f (x, reduce f e y);

(* esempio con errore sulla somma ( il + è infisso) *)
- reduce + 0 [1,2,3]; (* errore *)

(* esempio corretto sulla somma *)
- reduce (op +) 0 [1,2,3];
val int = 6 : int

(* esempio sulla media *)
- let
fun f L (elem, accum) = elem / real(lenght L) è accum
```

```
val lista = [1.0, 2.0, 3.0]
    in
reduce (f lista) 0.0 lista
    end;

val it = 2.0 : real
```

### 19.4 Funzioni anonime

Quando utilizziamo funzioni di ordine superiore, può far comodo passargli funzioni semplici. Queste funzioni anonime si specificano con la keyword `fn`

```
- fn x => x+1;
val it = fn : int -> int

(* per sommare uno a tutti gli elementi di una lista *)
- map (fn x => x+1) [1,2,3];
val it = [2,3,4] : int list
```

In Lisp e Scheme l'equivalente di *fn* è la keyword *lambda*:

- Storicamente deriva dal *lambda calcolo*, un modello di calcolo matematico basato su funzioni di ordine superiore a cui tutti i linguaggi funzionali si sono ispirati;
- Nel lamba calcolo l'operatore $\lambda$ è l'analogo di *fn*.

**NB**: Una funzione anonima non può essere ricorsiva perchè non esiste l'identificatore necessario alla chiamata della funzione

### 19.5 *Val* vs *fun*

```
- fun f x = x +1;
val f = fn : int -> int

- val f = fn x => x+1
val f = fn : int -> int
```

Pertanto da come si può vedere *fun* è il cosidetto zucchero sintattico, cioè una utile abbreviazione per qualcosa che si potrebbe fare in un altro modo (con *val*).

## 19.6 Ulteriori dettagli sul curryng

```
- fun f' x y = x+y
```

Questa funzione può essere definita come:

```
- fun f' x = fn y => x+y;
val f' = fn : int -> int -> int
```

È anche possibile definre funzioni che effettuano il curryng e la sua trasformazione inversa per una data funzione del tipo giusto:

```
(* f deve accettare una coppia (x,y) *)
val curry_2args = fn f => fn x => fn y => f (x,y)

(* f deve essere del tipo f x y *)

val uncurry_2args = fn f => fn (x,y) => f x y
```

Un pratico esempio:

```
fun f(x,y) = x+y

val f' = curry_2args f; (* equivalente a f' x y = x+y *)

fun f' x y = x+y;

val f = uncurry_2args f'; (* equivalente a f(x,y)= x+y *)
```

## 19.7 Polimorfismo parametrico (simil template)

Sono tipi di dato che non sono completamente specificati.

Un esempio è il caso del costruttore di liste `::` (può essere applicato a qualsiasi tipo).

La funzione *length* prende una lista di elementi il cui tipo `'a` non è specificato:

- Cioè accetta liste con qualsiasi contenuto (è un informazione non rilevante → deve soltanto contare i nodi).

Analogamente anche la funzione *map* è parametrica.

### 19.7.1 Come definire i tipi parametrici (con le liste)

```
(* generalizzazione delle nostre liste *)

- datatype 'a lista = vuota | nodo of ('a * 'a lista);

datatype 'a lista = nodo of 'a * 'a lista | vuota

(* generalizzazione degli alberi binari *)

- datatype 'a bt = emptybt | btnode of ('a * 'a bt * 'a bt);
datatype 'a bt = btnode of 'a * 'a bt * 'a bt | emptybt
```

Per le funzioni (quando il compilatore non ha bisogno di aiuto per stabilirne il tipo) non dobbiamo fare niente di speciale (ci pensa il type inference)

```
- fun conta(vuota) = 0 |
      conta (nodo (_,1)) = conta(1) + 1;

val conta  = fn : 'a lista -> int
```

ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*.

Nel caso in cui si vuole che il tipo supporti l'uguaglianza, allora si mette un doppio apice (`"a`).

## 19.8 Fattoriale in ML

```
fun fac 0 = 1
  | fac n = n * fac (n - 1)
```

# 20 Lezione del 28-05

## 20.1 Encapsulation e interface

Le *signature* sono il costrutto ML per definire interfacce. Definiscono tipi e funzioni senza specificare come sono implementati.

*es* STACK:

```
signature STACK =
```

```
sig
    type 'a stack
    val empty: 'a stack
    val push: ('a * 'a stack) -> 'a stack
    val pop: a' stack -> ('a * 'a stack)
end;
```

Dichiara:

- Un tipo parametrico 'a stack senza dire come è definito;
- Una funzione *empty* per costruire uno stack vuoto;
- Una funzione *push* per inserire un elemento nello stack;
- Una funzione *pop* per estrarre la testa dallo stack;

Il tutto senza ovviamente dire come sono implementate.

## 20.2 Structure (implementazione delle signature)

Le *structure*, come le classi, definiscono tipi di dato astratti

*es* STACK:

```
structure Stack :> STACK =
struct
      type 'a stack = 'a list;
      val empty: [];
      val push(x,s) = x :: s;
      val pop(x::s) = (x,s);
end;
```

Con l'espressione `Stack :> STACK` stiamo dicendo che:

1. Stack deve implementare tutti gli identificatori dichiarati in STACK;
2. I tipi di dato dichiarati in Stack possono essere utilizzati *solo* con le operazioni dichiaraye in STACK
    (a) Ogni altra funzione definita nella *structure* non è accessibile da fuori;
    (b) Si ottiene quindi l'*encapsulation*;
    (c) Si definiscono *tipi di dato astratti*.

## 20.3 Incapsulamento dell'implementazione dei tipi

Anche se in Stack il tipo stack è implementato con list

```
type 'a stack = 'a list;
```

Non si può usare come list, questo perchè la struttura Stack non offre alcuna funzione in più di quelle implementate e definite (quindi non è possibile utilizzare ad esempio la funzione *length*).

Si ha infatti non in una structure equivalence, ma in una name equivalence (stack nasconde l'implementazione del tipo)

## 20.4 Functors (structure parametrice)

Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri.

Si usa la keyword *functor* (analogo dei template).

```
functor Image( X : COLOR) =
  struct
    (* qui si può usare X come un tipo *)
    (* Con le oferazioni definite da COLOR *)
  end
(* Con il functor si possono generare diversi tipi di dato *)

structure Image_RGB = Image(RGB);
structure Image_CMYK = Image(CMYK);
```

## 20.5 Eccezioni e integrazione con type checking

Come Java, anche ML presenta le sue eccezioni predefinite:

```
- 3 div 0;
uncaught exception Div (divide by zero)
```

In ML le eccezioni vengono integrate con il type checking:

```
- fun pop(x::s) = (x,s);
Warning match nonexhaustive
x :: S => ...
```

```
- pop [];

uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. La type inference capisce che l'input di pop è una lista;
2. Il datatype lista ha 2 costruttori → `::` e `[]`;
3. La definizioni per casi di pop ne presenta una sola, da cui il warning;
4. Il compilatore inserisce automaticamente una eccezione Match nei casi mancanti.

Il programmatore ovviamente può definire le proprie eccezioni:

```
exception EmptyStack (* dichiara una nuova eccezione *)

fun pop(x::s) = (x,s) |
    pop [] = raise EmptyStack (* dove raise è il corrispettivo di throw *)
```

Il risultato in caso di errore è più esplicativo dell'eccezione automatica Match

Le eccezioni possono essere catturate e gestite con *handle*:

```
pop x
handle EmptyStack =>
  ( print "messaggio di errore specializzato";
  raise EmptyStack );
```

Il blocco *handle* può essere messo in qualsiasi blocco che possa generare una eccezione.

```
(3 div x) handle ...
```

Può gestire diverse eccezioni

```
<expression> handle
  <exception 1> => ... |
  <exception 2> => ... |
  ...
```

Ogni ordine è buono perchè non è un linguaggio OO (non c'è sovrapposizione → sono tutti oggetti diversi).

2 modi di usarlo in ML funzionale puro:

- Fare qualcosa come stampare un messaggio e *rilanciare l'eccezione*;
- Aggiustare l'errore restituendo un valore dello stesso tipo dell'espressione che ha sollevato l'eccezione.

Queste sono le uniche opzioni che passano il type checking senza errori.

Si possono inoltre aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni (il parametro viene letto con il pattern matching).

## 20.6 Esempio di un semplice compilatore

### 20.6.1 Definizione di un albero sintattico

```
datatype syntree = co of int (* le costanti *)

    | plus of syntree * syntree
    | minus of syntree * syntree
    | times of syntree * syntree
    | divide of syntree * syntree
    | modulus of syntree * syntree
```

### 20.6.2 Definizione del linguaggio target

Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente:

```
datatype instruction
    = LOADC of int * int
    = LOADI of int * int
    | STOREI of int * int
    | INCR of int
    | DECR of int
    | SUM of int * int
    | SUB of int * int
    | MUL of int * int
    | DIV of int * int
    | MOD of int * int
```

Dove:

- LOADC i c => Ri := c;
- LOADI i j => Ri := mem(Rj);
- STOREI i j => mem(Rj) := Ri;
- INCR i => Ri := Ri + 1;
- DECR i => Ri := Ri - 1;
- SUM i j => Ri := Ri + Rj;
- SUB i j => Ri := Ri - Rj;
- MUL i j => Ri := Ri * Rj;
- DIV i j => Ri := Ri / Rj;
- MOD i j => Ri := Ri mod Rj;

Il codice oggetto utilizza 2 registri:

- Il registro R1 come puntatore alla testa dello stack;
- Il registro R2 come accumulatore (per calcolare le singole operazioni)

### 20.6.3 La traduzione

La traduzione vera e propria è effettuata da una funzione ausiliaria *translate* che prende in input:

- Un albero sintattico *tree*;
- Una *continuazione*, ovvero il codice da eseguire dopo aver eseguito le operazioni contenute in *tree*.

Pertanto la prima chiamata a *translate* gli passerà:

- L'albero sintattico dell'intera espressione da compilare;
- La lista di istruzioni `[HALT]`.

### 20.6.4 Generazione del codice

```
fun codegen tree =
  let
      (* definizione della funzione translate *)
      (* R1: Stack Pointer; R2: accumulator *)
    fun translate(co, x) cont =
      LOADC(2,x) :: INCR(i) :: STORE(2,1) :: cont
      | translate(plus(t1,t2)) cont =
  translate t1 (
```

```
    translate t2(LOADC(2,1) :: DECR(1) :: LOADI(3,1) :: SUM(2,3) :: STOREI(2,1) :: cont))
  ...
  in
   translate tree [HALT]
end
```

Per il resto delle operazioni invece cambia poco (sono solo le operazioni a cambiare all'interno).

### 20.6.5 Ottimizzazione del codice - funzione optimize

La funzione *optimize* elimina le più comuni operazioni ridondanti.

Itera una funzione ausiliaria *opt1* che esegue un singolo passo di ottimizzazione.

Questo può attivare ulteriori semplificazioni → *optimize* itera *opt1* finchè il codice non può essere ulteriormente ridotto.

Per farlo utilizziamo la keyword *local*:

```
local
(* definizione singolo passo di ottimizzazione *)
fun opt1 [] = []
    | opt1(INCR(1)::STORE(y,z)::DECR(1)::cont) =
      let val cont' = opt1 cont in
if z=1 then cont'
else if y <> 1
    then STOREI(y,z)::cont'
else INCR(x)::STOREI(y,z)::DECR(x1)::cont'
end
| opt1 (STORE(x,y)::LOADI(x1,y1)::cont)=
  let val cont' = opt1 cont in
    if x=x1 andalso y=y1
      then STOREI(x,y)::cont'
      else STOREI(x,y)::LOADI(x1,y1)::cont'
end
| opt1 (c :: cont) = c :: (opt1 cont)
in
  fun optimize code =
      let val code' = opt1 code in (* fa 1 passo di ottimizzazione *)
if length(code') = length(code) ( nessun progresso )
    then code'
```

```
      else optimize code'
    end
end
```

### 20.6.6 Combinare le fasi con composizione di funzione

L'operatore `o` denota la composizione di funzioni:

$$(f \circ g)(x) = f(g(x))$$

Con la composizione è facile definire l'intero processo di compilazione assemblando le diverse fasi:

```
- val compile = optimize o codegen o parse;

val it = fn : string -> instruction list
```

La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare:

- In Java, ogni nodo dell'albero sintattico sarebbe un oggetto e *leggere* la struttura di pezzi di albero non sarebbe immediato;
- Inoltre la type inference ci permette di omettere il tipo degli identificatori, producendo un codice più snello
  - Come fosse uno scripting language debolmente tipato
  - Ma senza sacrificarne il controllo di tipi forte.

Per questi motivi linguaggi come ML vengono utilizzati per la prototipizzazione rapidaa di compilatori e interpreti.

## 20.7 Linguaggi logici

In questo tipo di linguaggi:

programmi → insiemi di assiomi

computazioni → dimostrazioni costruttive di una formula logica data (detta *query*) mediante gli assiomi del programma.

### 20.7.1 Implementazione e Interazione in Prolog

Simili a quelle di ML. L'implementazione è mista:

- I programmi Prolog vengono compilati in un bytecode;
- Questo viene interpretato da una vm (La Warren abstract machine (WAM)).

L'interazione con Prolog è analoga a quella con ML:

- Si inviano query all'interprete e si ottengono le relative risposte;
- Se il programma è stand alone, si interagisce mediante la sua UI.

### 20.7.2 Sistema consigliato per il corso

SWI Prolog (free):

- Implementa il Prolog standard;
- Supporta sia intepretazione che compilazione stand-alone.

Per invocarlo: `swipl`

`?-` è il prompt dell'interprete.

Per caricare un proprio programma `mioprog.pl`:

```
?- ['mioprog.pl'].
```

```
?- consult('mioprog.pl').
```

```
?- reconsult('mioprog.pl').
```

Da notare:

- Il nome del file viene messo tra singoli apici perchè in prologo esiste l'operatore `.` (dot);
- Con la key `consult` si caricara il programma
- Con la key `reconsult` si ricaricara il programma in seguito ad una correzione (questo per evitare interferenze).

### 20.7.3 Costrutti base

Tre tipi di statement:

- Fatti (facts);
- Regole (rules);
- Queries (detti anche goals);

Presenta una sola struttura dati → termini logici (logical terms).

Fatti

Asseriscono una relazione tra oggetti

```
father(abraham, isaac). (abraham è padre di isacco)
```

*father* oltre che relazione, è chiamato anche *predicato*.

**NB**: I nomi dei predicati *devono* iniziare per lettera minuscola.

Gli argomenti `abraham, isacc` iniziano con lettera minuscola, in quanto sono costanti (come i numeri). Se iniziassero con la lettera maiuscola sarebbero variabili.

Con i fatti possiamo definire un *database*.

Ogni predicato corrisponde a una *tabella relazionale*

Queries

I programmi logici sono fatti per rispondere a *queries*.

Le query hanno la stessa forma dei fatti. Sono entrambi dei cosidetti atomi logici:

- Nei programmi sono asserzioni;
- Nelle query sono domande.

NB: Nel libro le query sono indicate con un punto interrogativo come domande.

### 20.7.4 Variabili logiche nelle query

Si riconoscono perchè iniziano con una lettera maiuscola:

```
?- father(abraham, X) /* esiste X tale che father(abraham, X)? */
X=isaac.
```

La vm cerca i valori che - sostituiti a X - rendono la query uguale a uno dei fatti nel programma.

**NB**: questo vale per i programmi di soli fatti.

### 20.7.5 Variabili logiche in generale

Differenza tra le variabili logiche e le variabili logiche di altri paradigmi:

- Le variabili logiche rappresentano *oggetti qualsiasi*, non specificati;
- Le variabili dei linguaggi imperativi sono locazioni di memoria;
- Gli identificatori dei linguaggi funzionali denotano valori immutabili.

Si possono usare anche nei fatti:

```
creato_da(X, dio) /* Ogni cosa è creata da dio */
```

Differenza tra le variabili nei fatti e nelle query:

- Nelle query → esistenzalmente quantificate;
- Nei fatti → universalmente quantificate;

### 20.7.6 I termini

Sono l'unica struttura dati in Prolog.

Si costruiscono con costanti, variabili (logiche) e *funtori* (che si comportano come i costruttori di ML, e che sono caratterizzati da nome e *arietà* (numero di argomenti)):

```
<term> ::= <constant> | <variable> |
<functor name> '(' [<term> [ ',' <term>]*] ')'
```

*es.*

```
successor(Int)
date(25, april, 2020)
color(rgb, 0,0,1)
```

Non ci sono dichiarazioni di tipo.

Costanti simboliche e funtori si usano senza dichiararli prima.

Gli argomenti di un predicato possono essere termini qualsiasi:

```
born(john, date(10, october, 2020))
```

Pertanto la sintatissi dei fatti è:

```
<fact> :: <atomic formula> '.'

<atomic formula> ::=
    <predicate> '(' [<term> [ ',' <term>]*] ')'
```

(Notare il punto dopo il fatto)

Esempi di query a un programma:

```
?- born(X, date(Y, october, 2000))
X=john
Y=10
```

La stessa variabile $X$ può comparire più volte nello stesso fatto o nella stessa query.

Significa che i termini nelle posizioni dove si trova $X$ devono essere uguali tra loro

### 20.7.7 Differenze tra termini di Prolog e pattern di ML

In ML non posso usare la stessa variabile più volte nello stesso patern (perchè servono solo ad estrarre informazioni da una struttura, non a controllare che elementi diversi siano uguali).

### 20.7.8 Ground e nonground

Un termine è *ground* se non contiene variabili. In caso contrario è *nonground*.

```
foo(a,b) /* ground */

bar(A) /* nonground */
```

Gli stessi aggettivi e gli stessi criteri si applicano ai fatti, alle query e anche alle regole

```
father(abraham, isaac) /* ground */

sum(X,0,X) /* nonground */
```

### 20.7.9 Sostituzione

Una *sostituzione* è un insieme finito di coppie:

$$\Theta = \{\mathrm{X_1} = \mathrm{t_1}\ ,\ldots,\ \mathrm{X_n} = \mathrm{t_n}\}$$

Dove:

- Le $X_i$ sono variabili e i $t_i$ termini;
- Le $X_i$ sono tutte diverse tra loro;
- Nessuna delle $X_i$ compare dentro i $t_i$.

L'applicazione di $\Theta$ a una espressione $E$ (che potrebbe essere termine, un fatto, una query o una regola):

- Si denota con $E\ \Theta$;
- Sostituisce le occorrenze delle variabili $X_i$ in $E$ con i rispettivi termini $t_i$.

*es.*

Se $\Theta = \{\ X = \text{isaac}\ \}$ e $E = \text{father(abraham, X)}$ allora:

$$E\ \Theta = \text{father(abraham,isaac)}$$

### 20.7.10 Istanze

L'applicazione di una sostituzione $\Theta$ a $E$ crea un caso particolare di $E$ dove le variabili di $E$ (che indicano oggetti non specificati) vengono sostituite con valori specifici (termini ground) o parzialmente specificati (termini nonground).

$E_1$ è un'istanza di $E_2$ se esiste $\Theta$ tale che $E_1 = E_2\ \Theta$, cioè se $E_1$ è un caso particolare di $E_2$ dove alcune variabili di $E_2$ sono istanziate, cioè legate a un valore

### 20.7.11 Unificazione

L'algoritmo di unficazione è quello che effettua il matching.

Prende due espressioni $E_1$ ed $E_2$ e (se possibile) restituisce una sostituzione $\Theta$ tale che:

$$E_1\ \Theta = E_2\ \Theta$$

detto *unificatore* (unifier).

L'unificatore costruito dall'algoritmo viene detto *most general unifier* (mgu) perchè non sostituisce una variabile con un termone se non necessario:

- Vincola il meno possibile il risultato $E_1\ \Theta$, lasciando le variabili libere quando può;
- Tecnicamente ogni altro unificatore $\Theta^1$ porta ad un istanza (caso particolare) di $E_1\ \Theta$ cioè
  - esiste una sostituzione $\sigma$ tale che $(E_1\ \Theta)\ \sigma = E_1\ \Theta^1$

*es.*

$$\text{mgu(sum(A,B,C), sum(X,0,X)) = \{A=X, B=0, C = X\}}$$

$$\text{mgu(sum(0,X,0), sum(Y,0,Y)) = \{X=0, Y=0\}}$$

$$\text{mgu(sum(0,X,0), sum(Y,Z,1)) } \rightarrow \text{ ERRORE}$$

Non esiste a causa del terzo argomento (non si può rendere 0 uguale a 1).

### 20.7.12 Costruzione delle risposte da soli fatti

Per trovare una risposta a una query del tipo $q(u_1, \ldots, u_m)$ si cerca nel programma il primo fatto $p(u_1, \ldots, u_m)$ con lo stesso funtore (cioè $p = q$ e $m = n$) e se ne trova uno:

- Calcola $\Theta = mgu(q(t_i, \ldots, t_n), p(u_i, \ldots, u_m))$

Se $\Theta$ esiste, allora:

- Si restituisce $\Theta$ come risposta (se vuota Prolog restituisce *true*);

- Poi se l'utente non vuole altre soluzioni si termina.

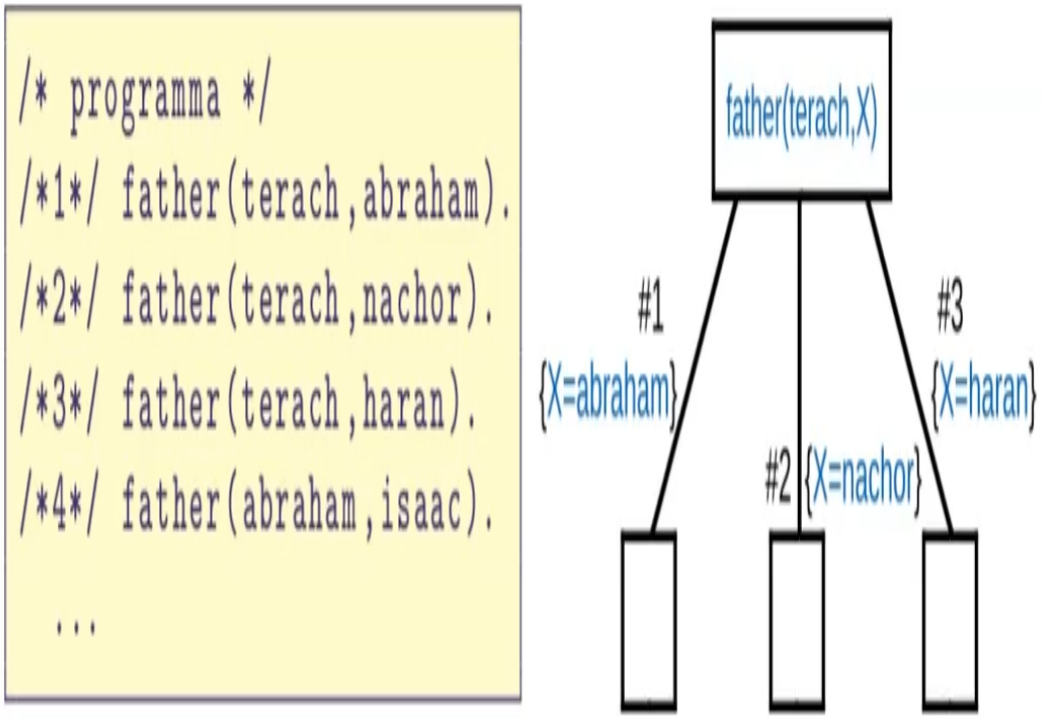
In caso contrario si cerca il prossimo fatto con lo stesso funtore.

Se esiste si ricalcola $\Theta$, altrimenti si termina.

Prolog restituisce *false* quando nessun fatto unifica con la query.

### 20.7.13 Rappresentazione grafica del procedimento

Un *search tree* per la query `father(terach, X)`:




Gli archi corrispondono ai fatti che unificano con la query.

Sono etichettati con:

- Il numero del fatto utilizzato (nell'ordine in cui compare nel programma);
- Il mgu della query e del fatto.

Il *search tree* di una query (rispetto a un programma) comprende tutte le risposte che Prolog genera se l'utente glielo chiede.

# 21 Lezione del 01-06

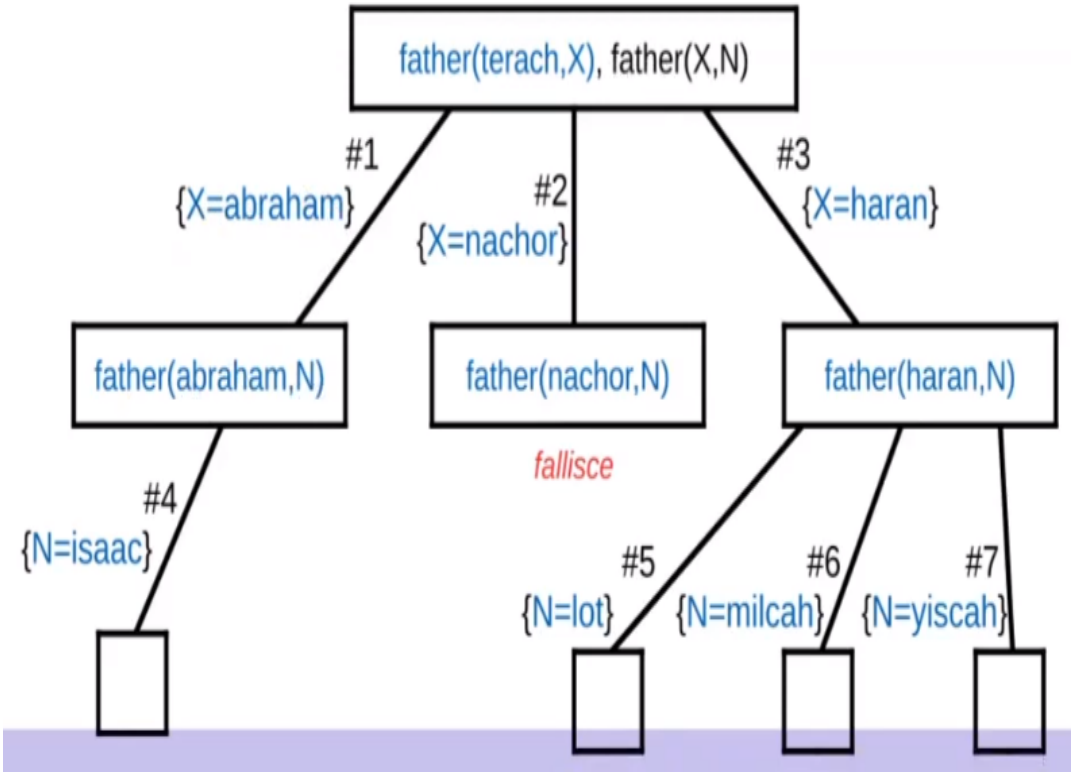
## 21.1 Conjunctive Queries

Le query possono contenere più formule atomiche (*goals*).

*es.*

```
father(terach.X),father(X.Nipote)
```

Dove la `virgola` è come un *and*, un *join*




(Prolog cerca di risolvere l'albero da sinistra verso destra)

## 21.2 Le Regole

*X è figlia di Y se Y è padre di X e X è femmina* in prolog corrisponde a:

```
daughter(X,Y) :- father(Y,X), female(X).
```

In pseudo-codice

```
<rule> ::= <atomic formula> [:- <goal list>].

<goal list> ::= <atomic formula> [, <atomic formula>]*
```

La formula atomica prima di `:-` è detta testa (head), mentre la parte successiva è detta corpo (body).

I fatti pertanto non sono che regole senza corpo.

Un programma logico è un insieme di regole.

### 21.2.1 Ragionamento

Si cerca una testa che unifica con la query e sostituisce la query con il corpo istanziato con il mgu.

Bisogna fare attenzione con le variabili → quelle della query devono sempre essere diverse da quelle della regola.

Per evitare questo problema, ogni volta che una regola viene usata la WAM ridenomina le sue variabili:

```
daughter(_101,_102) :- father(_102,_101), female(_101).
```

Per evitare di perdere le figlie delle donne aggiungiamo una seconda regila:

```
daughter(X,Y) :- mother(Y,X), female(X).
```

Possiamo inoltre introdurre il concetto di genitore per semplificare le 2 regole:

```
parent(X,Y) :- father(Y,X), female(X).
parent(X,Y) :- mother(Y,X), female(X).
```

## 21.3 Overloading

Si può usare lo stesso npome di predicato con numeri diversi di argomenti.

Simili predicati vengono trattati come predicati diversi.

## 21.4 Wildcards

Prolog supporta le wildcards:

```
mother(Mom) := mother(Mom,_).
```

**NB**: Se usiamo 2 volte una variabile allora in quei punti devo avere lo stesso valore, mentre ogni wildcard può essere associata a un valore diverso:

```
?- mother(X,X).
false (nessuna è madre di sè stessa)

?- mother(_,_).
true (cerca nel DB un fatto mother(X,Y))
```

### 21.5 Esempio di search tree per ancestor

```
ancestor(X,Y) := parent(X,Y).
ancestor(X,Y) := parent(X,Z), ancestor(Z,Y).
```

### 21.6 Prolog e l'algebra relazionale

Ogni tabella relazionale `r` si può rappresentare con dei fatti

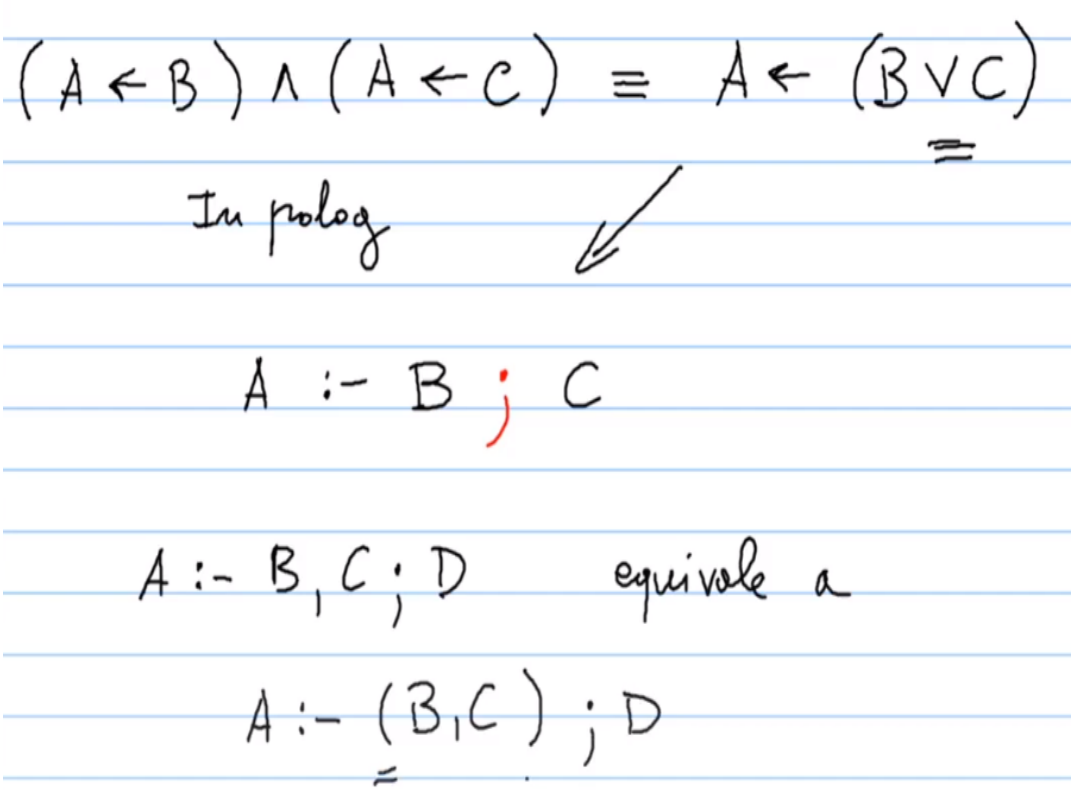
| | | | |
|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ |
| $a_2$ | $b_2$ | $c_2$ | $d_2$ |
| $a_3$ | $b_3$ | $c_3$ | $d_3$ |
| $a_4$ | $b_4$ | $c_4$ | $d_4$ |
| ... | ... | ... | ... |

$r(a_1, b_1, c_1, d_1)$

$r(a_2, b_2, c_2, d_2)$

$r(a_3, b_3, c_3, d_3)$

$r(a_4, b_4, c_4, d_4)$




Di seguito alcuni esempi di operazioni in prolog:

```
/* UNIONE di r e s */
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).

/* INTERSEZIONE di r e s */
r_inters_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).

/* PROIEZIONE di r, ad es. su colonne 1 e 3 */
r13(X1,X3) :- r(X1,...,Xn).

/* SELEZIONE di r */
r_sel(X1,...,Xn) :- r(X1,...,Xn), <condizione>.

/* ad esempio: */
r_with_2_less_than_3(X1,...,Xn) :- r(X1,...,Xn), X2 < X3.
```

Per la differenza tra relazioni occorre un operatore di negazione denotato da `\+`

La negazione trasforma false in true → fallimenti in successi.

```
r_minus_s(X1,...,Xn) := r(X1,...,Xn), \+ s(X1,...,Xn)
```

L'albero deve essere finito. Se un programma cade in una ricorsione infinita ovviamente non termina

### 21.7 Liste

Analoga ad ML:

- Costruttore lista vuota → `[]`;
- Costruttore nodi → `[elem|resto]`;

Notazioni alternative equivalenti:

| Abbreviata | Costruttori espliciti |
|---|---|
| `[a]` | `[a I[]]` |
| `[a,b]` | `[a I [b I []] ]` |
| `[a,b,c]` | `[a I [b I [c I [] ]] ]` |
| `[a I X]` | `[a I X ]` |
| `[a,b I X]` | `[a I [b I X] ]` |

È possibile esprimere liste *parzialmente specificate*, dove alcuni elementi ed eventualmente la coda sono variabili:

```
[a, X, b | Y]
```

### 21.7.1 Predicato `member`

Cerca un elemento $X$ in una lista $L$:

- È ricorsivo;
- Si usano le wildcard.

```
member(X, [ X | _]).
member(X, [ _ | L]) :- member(X, L).
```

Somiglia alla definizione per casi in ML, ma in prolog posso usare la stessa variabile più volte per esprimere pattern dove certi elementi sono uguali.

### 21.7.2 Evanescenza di parametri di Input e Output

Non è presente una chiara distinzione tra input e output:

- Ogni parametro può essere legato a un termine con costruttori (input);
- Ogni parametro attuale con una variabile libera produce delle sostituzioni (output).
- I parametri con almeno un costruttore e una variabile fornisco sia un input sia un output.

Con **member**:

- `member(X, <lista ground>)` prende una lista e ne restituisce i suoi elementi → Modalità `IN/OUT`;
- `member(<elem ground>, L)` prende un elemento e ne restituisce le liste che lo contengono → Modalità `IN/OUT`;
- **Invertibilità dei predicati** → possono rappresentare sia una funzione sia la sua inversa.

Incontreremo lo stesso fenomeno in `append/3` (predicato *append* con 3 argomenti).

# 22 Lezione del 04-06

## 22.1 Il predicato *append*

Questo predicato prende in input 2 Liste e le concatena nella lista finale:

`append(Lista1, Lista2, Risultato)`

Caso base: `append([], L,L)`

```
append([X|Tail], L, [X|RisParz] ) :- append(Tail, L, RisParz).
```

Nel secondo caso va effettuata la chiamata ricorsiva per concatenare `Tail` ed `L`

## 22.2 Generazione di tutti i `prefissi/suffissi` di una lista

```
prefix(Prefix, List) :- append(Prefix,_, List).

suffix(Suffix, List) :- append(_,Suffix, List).
```

Con member:

```
member(X,L) :- append(_, [X|_], L).
```

## 22.3 Calcolo delle sottoliste di L

S è una sottolista di L se valgono queste 2 condizioni (equivalenti):

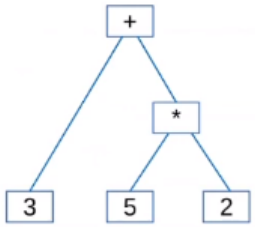
- S è il *prefisso* di un *suffisso* di L
- S è il *suffisso* di un *prefisso* di L

```
sublist(Sub, L) :- prefix(Pre, List), suffix(Sub, Pre).
```

## 22.4 Calcolo simbolico delle derivate

In prolog gli operatori aritmetici sono costruttori:

- 3+5*2 *non* è uguale a 13!
- denota invece l'albero sintattico qui a destra
- Per calcolare 3+5*2 usare is:
  `Ris is 3+5*2`
- questo goal è vero se Ris è uguale al *valore* di 3+5*2




### 22.4.1 Schema del predicato

`der(Expr, X, D)` è vero se D è la derivata di Expr rispetto a X:

*es.*

```
der(X, X, 1).

der(X^N, X, N * X^N1) :- N1 is N-1
```

Si possono aggiungere predicati per semplificare il risultato.

## 22.5 Programmazione non deterministica

Invece di dire a Prolog come trovare la soluzione di un problema si dice cosa sia una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree).

Schema generale

```
solution(X) :- generale(X), test(X).
```

Prolog genera via via i candidati a soluzione X.

Per ciascuno di essi verifica se è effettivamente una soluzione (test).

Se si restituisce la soluzione. In caso contrario fa backtrack e torna a generare il successivo candidato.

Se si chiedono altre risposte, genera altre soluzioni.

Questo tipo di programmazione è una caratteristica unica del paradigma logico.

```
// paradigma imperativo //

X := primo candidato;
    while not test(X) and X != null do
X := prossimo_candidato(X)
    return X

// paradigma logico //
fun soluzione(X) =
  if test(X) then X
    else soluzione(prossimo_candidato(X))

// Per invocarlo

soluzione(primo_candidato)
```

In entrambi i casi verrebbe generata solo la prima soluzione trovata ( se si vogliono le altre occorre complicare il gioco).

## 22.6 Ricerca dei membri pari di una lista

```
/* stile generate and test */
membro_pari(X,L) :- member(X,L), 0 is X mod 2.

/* invece di ricorsione ad hoc */
membro_pari(X,[X|_]) :- 0 is X mod 2.
membro_pari(X,[_|Resto]) :- membro_pari(X,Resto).
```

Notare come l'approccio `generate & test` possa giocare il ruolo delle funzioni di ordine superiore in ML:

- Permette di comporre facilmente nuovi predicati da quelli dati;
- In questo caso member, che diventa uno strumento generale per visitare una lista, funge da *filter* (query congiuntiva).

## 22.7 Compilazione stand-alone di componente grafica

Se si usa `@` (per generare un goal) bisogna dichiararlo come operatore unario prefisso (altrimenti si va in contro ad un syntax error)

```
/* Inserire all'inizio del programma */

:- op(1, fx, @).
```

## 22.8 Caratteristiche uniche del prolog

### 22.8.1 Invertibilità dei predicati

- Un singolo predicato implementa molte funzioni;
- Dovuto al fatto che le variabili logiche sono `IN/OUT/IN-OUT` a seconda dei parametri attuali.

### 22.8.2 Programmazione non deterministica

- Con ricerca automatica delle soluzioni (ci si basa sul backtracking → meccanismo di visita del search tree).

# 23 Lezione del 08-06 - lezione finale (Esericitazione)

Esercitazione