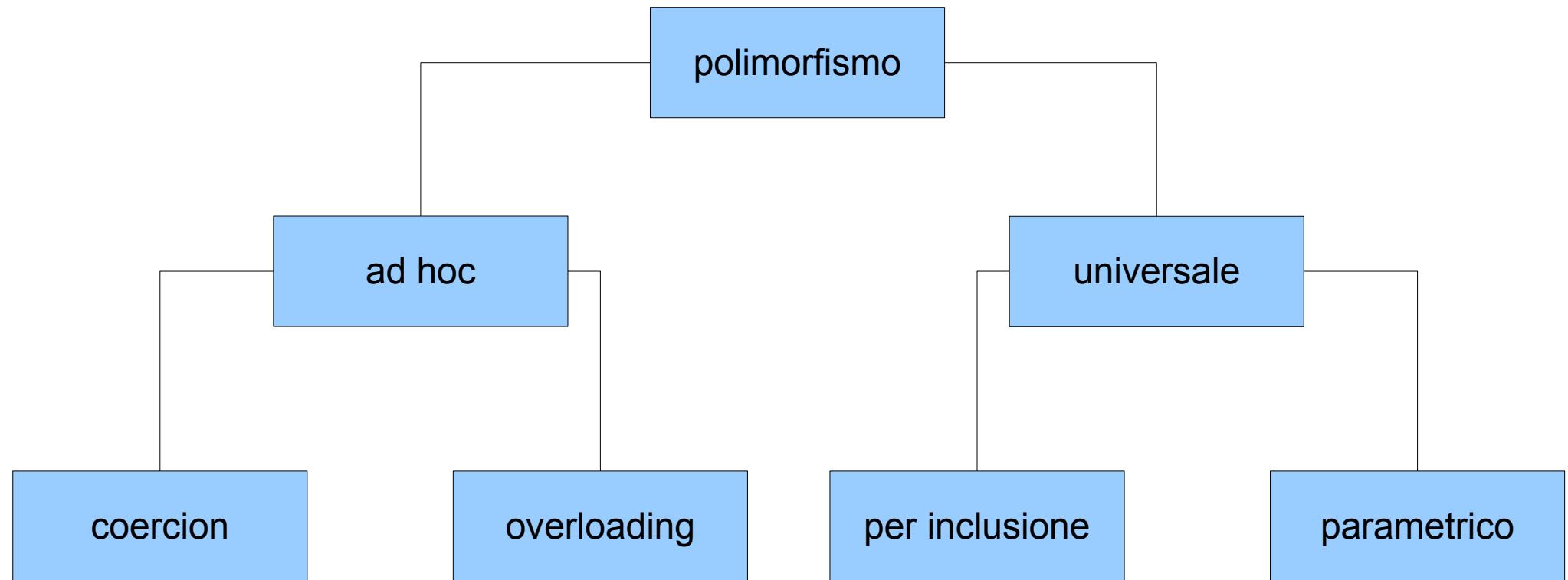


Polimorfismo

- Chiamate di funzioni/procedure/metodi con
 - Sintassi simile
 - Significato (o comportamento) diversi a seconda del contesto
- Può essere definito in vari modi

Polimorfismo



Polimorfismo

- Ad hoc
 - specifico per particolari tipi di dato
 - compresi alcuni predefiniti
 - con i meccanismi ad hoc si aggiungono singoli casi di polimorfismo ogni volta
- Universale
 - si applica ad un numero di casi illimitato a priori, ad esempio:
 - aggiungendo una nuova superclasse con un metodo concreto, l'overriding crea casi di polimorfismo rispetto a tutte le sottoclassi che già definiscono quel metodo
 - un template può definire una varietà di metodi di tipo diverso (si veda l'esempio delle liste) il cui numero non è limitato a priori

Polimorfismo ad hoc

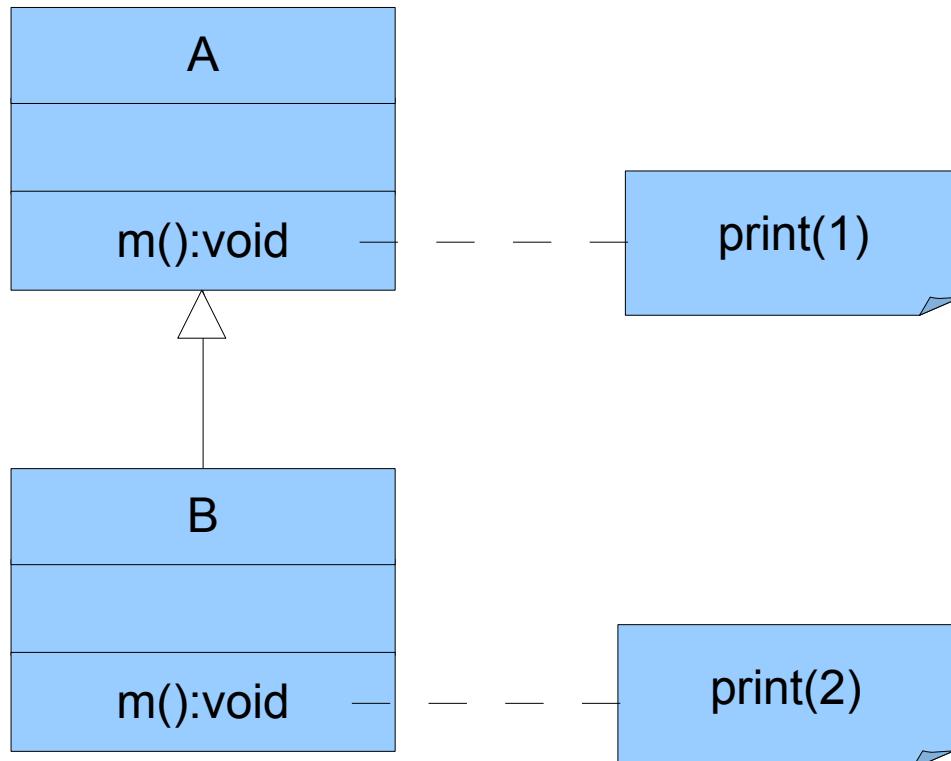
- Overloading: *stesso nome ma diversa implementazione* a seconda dei tipi dei parametri
 - $23 + 4 : (\text{int}, \text{int}) : \text{int}$
 - $12.34 + 1.0 : (\text{float}, \text{float}) : \text{float}$
 - (internamente implementate con istruzioni diverse)
 - “Abc” + “dE3” : $(\text{string}, \text{string}) : \text{string}$
 - (concatenazione)

Polimorfismo ad hoc

- Coercion: promozione automatica di tipi
 - $12.34 + 1$
 - $\langle \text{float} \rangle + \langle \text{int} \rangle$
 - converte $\langle \text{int} \rangle$ a $\langle \text{float} \rangle$
 - converte 1 in 1.0
 - si riduce a $\langle \text{float} \rangle + \langle \text{float} \rangle$
 - apparentemente + ha *anche* il tipo $(\text{float}, \text{int}): \text{float}$
 - nonchè $(\text{int}, \text{float}): \text{float}$
 - diverse implementazioni (le conversioni implicite fanno parte dell'implementazione dell'istruzione)

Polimorfismo universale

- Per inclusione
 - overriding nelle sottoclassi



A x;
x = new A();
x.m(); 1

x = new B();
x.m(); 2

Polimorfismo universale

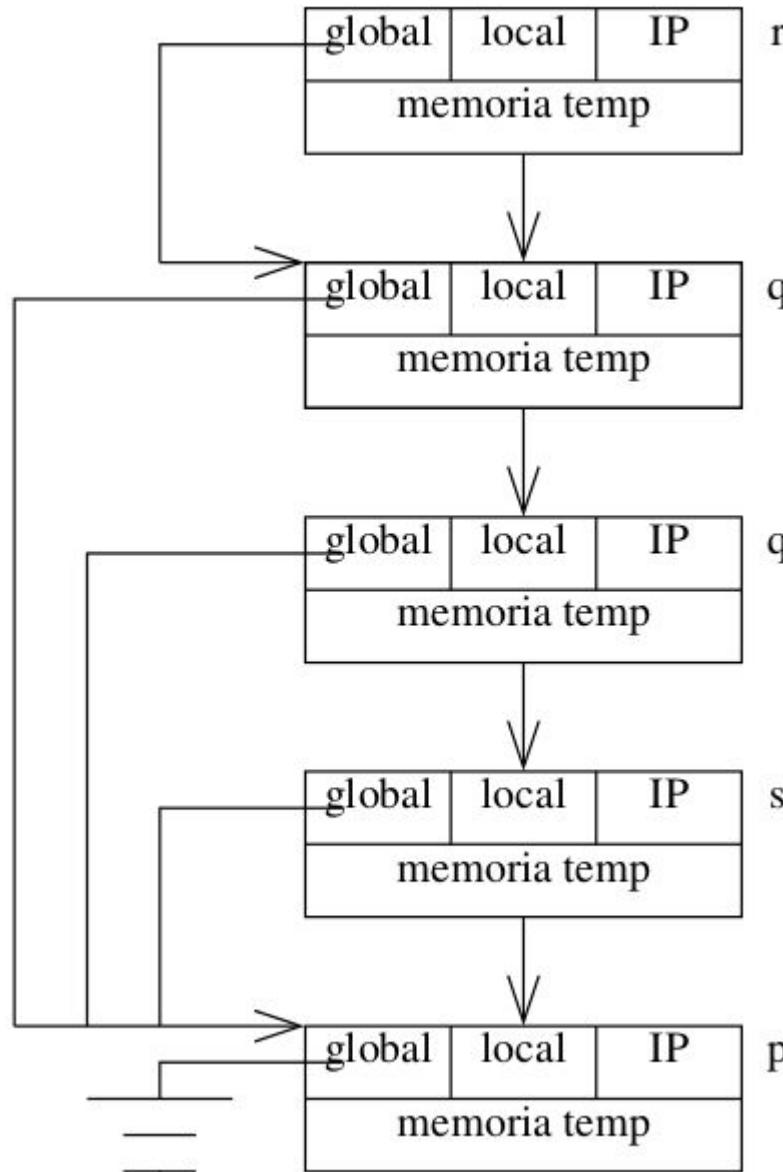
- Parametrico
 - *tipi parametrici* o *templates*
 - adatto a definire strutture omogenee
 - ma si vedano le considerazioni sugli approcci ibridi nelle slide di confronto tra polimorfismo per inclusione e parametrico

Ambiente non locale

dal paradigma imperativo a quello a oggetti

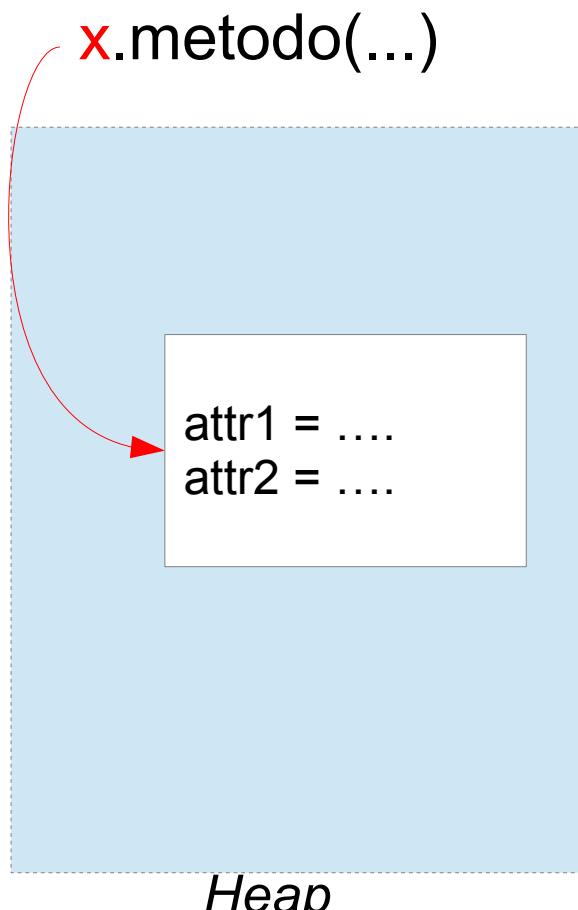
Piero Bonatti

Ambiente non locale nel paradigma imperativo



- Interamente contenuto nello stack di attivazione
- Interamente determinato dallo scoping
 - (statico o dinamico)

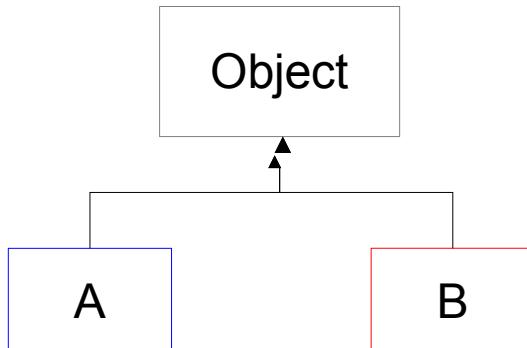
Ambiente non locale nel paradigma a oggetti – versione base



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

Ambiente non locale nel paradigma a oggetti – versione generale

```
class A {  
    type1 attr1;  
    class B {  
        type2 attr2;  
        metodo(...)  
    }  
}
```

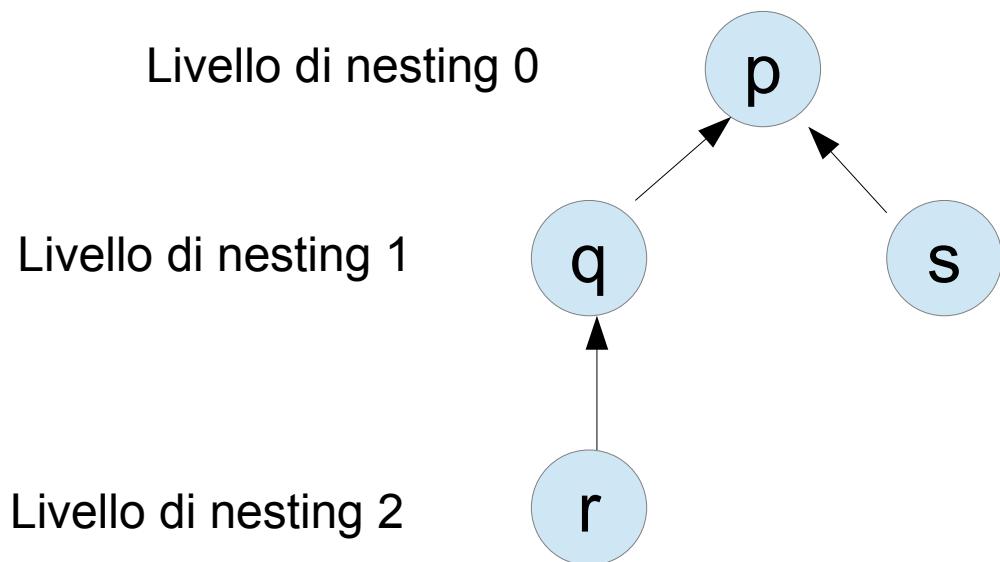


- Ancora scoping statico
 - (attributi dichiarati nei blocchi-classe contenenti il metodo)
- È ancora allocato nello heap
 - sono attributi di *oggetti*
- Ma uscendo da ogni blocco si cambia classe
 - passando quindi a un oggetto diverso!
 - **Quale???**
- Stessa filosofia: Viene indicato esplicitamente
 - **Come???**

Annidamento (nesting)

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

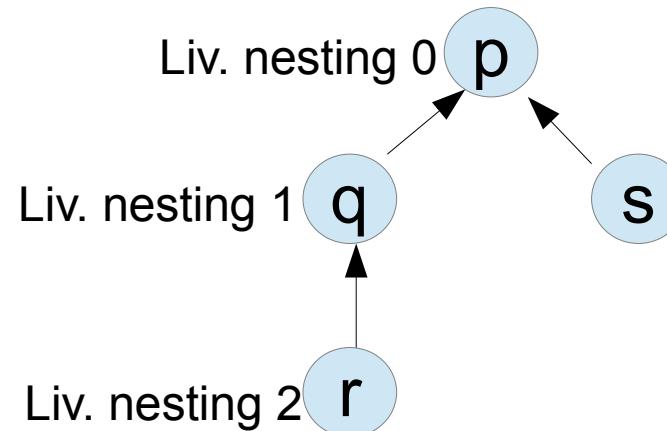
- L'annidamento si può rappresentare come un albero
- I livelli dell'albero di nesting corrispondono ai livelli di nesting



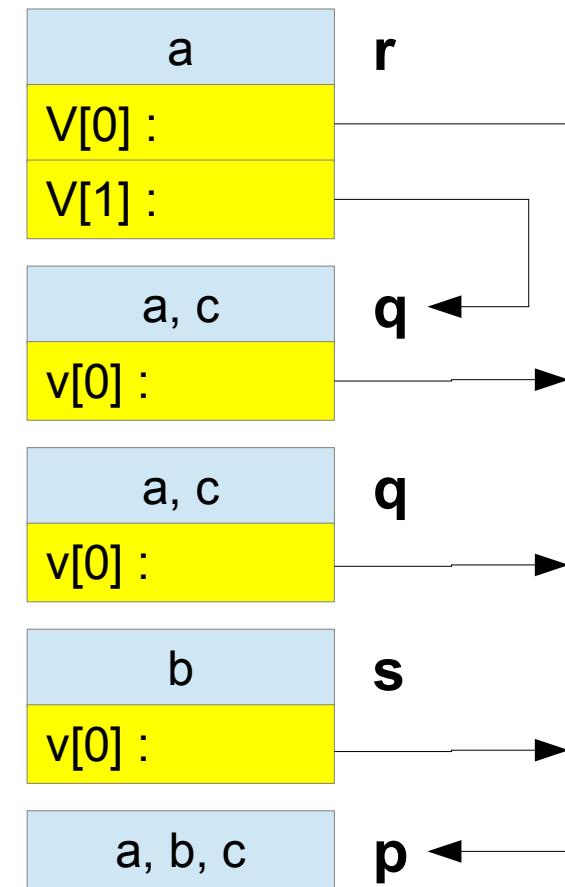
Le frecce indicano l'ambiente non locale

Rappresentazione variabili 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;  
...
```



Stack di attivazione



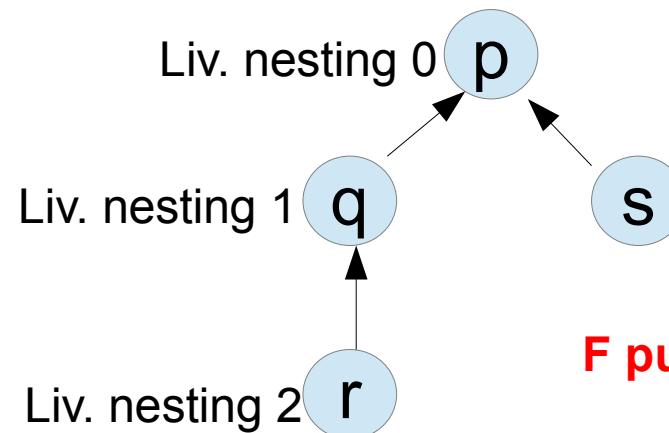
Rappresentazione variabili non locali:
(livello, offset)

Nella procedura **r**:
b è rappresentata da **(0,2)**
c è rappresentata da **(1,2)**

env(x) = v[livello] + offset

Visibilità delle procedure

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



(Ambito statico)

F può chiamare G se:

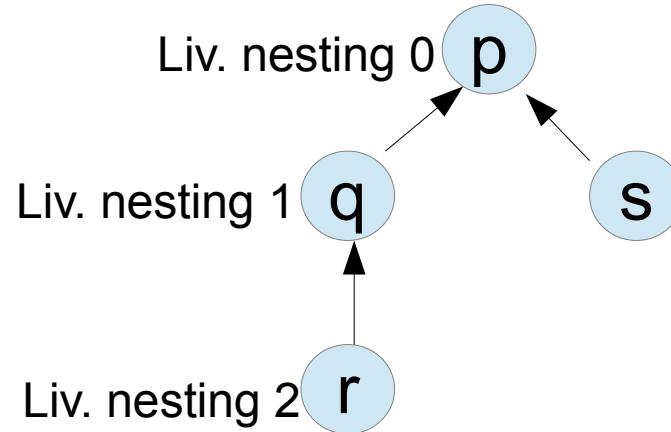
- G è definita in F
- G è definita in uno dei blocchi che contiene F

quindi:

F e G hanno sempre una parte di ambiente non locale in comune

Mantenimento del vettore degli ambienti non locali

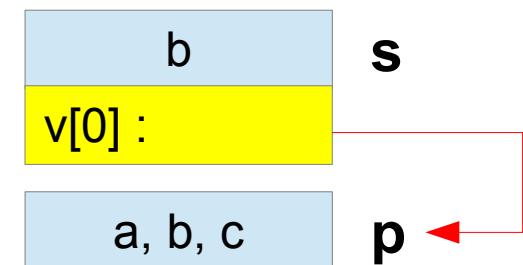
```
program p;  
var a,b,c int;  
  
procedure q;  
var a,c int;  
  
procedure r;  
var a int;  
...  
...  
  
procedure s;  
var b int;  
...  
...
```



Chiamata a procedure definite localmente:

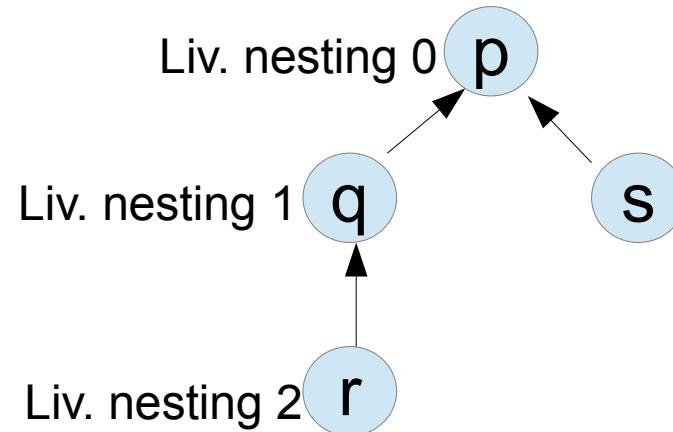
Esempio 1: p chiama s:

- Aumenta liv. di nesting
- Aggiungere elem. a v[]



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;  
...  
...  
...
```

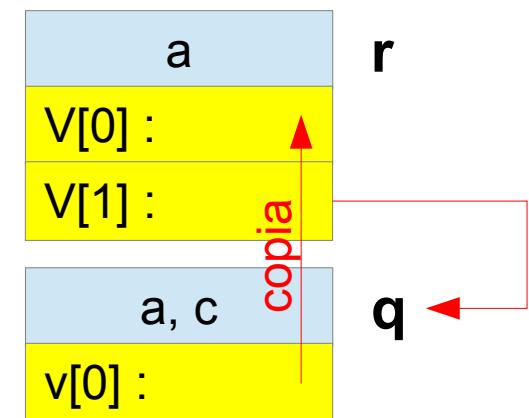


Chiamata a procedure definite localmente:

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

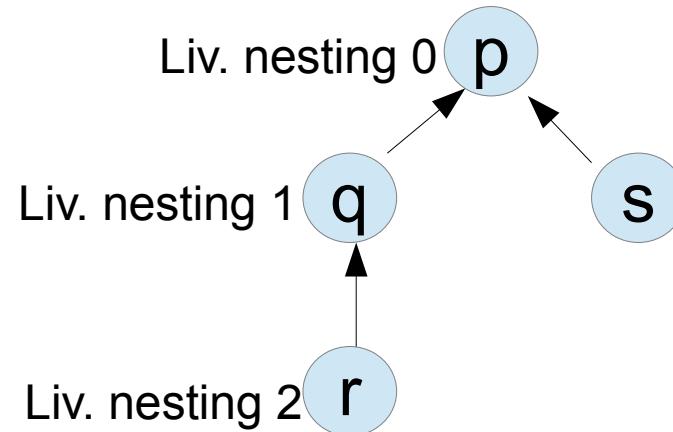
Esempio 2: q chiama r:

- Aumenta liv. di nesting
- Aggiungere elem. a v[]
- Copiare gli altri elementi



Mantenimento del vettore degli ambienti non locali

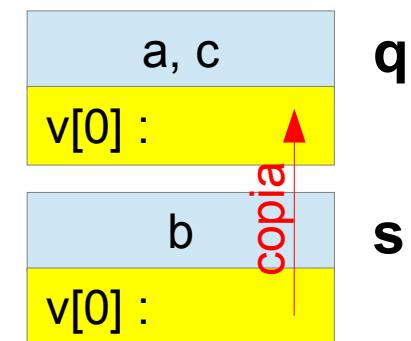
```
program p;  
var a,b,c int;  
  
procedure q;  
var a,c int;  
  
procedure r;  
var a int;  
...  
...  
  
procedure s;  
var b int;  
...  
...
```



Chiamata a procedure definite **esternamente**:

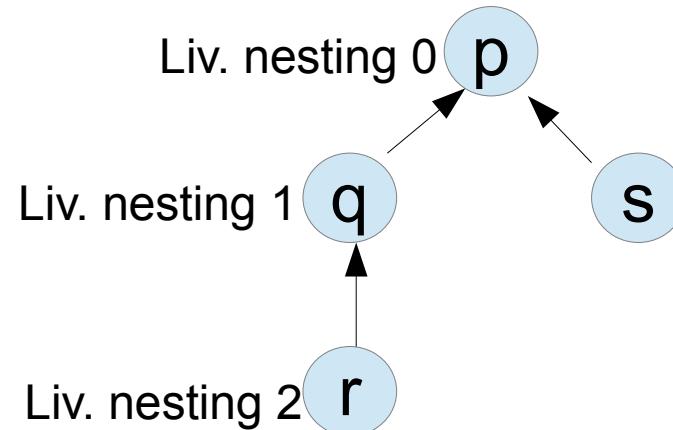
Esempio 1: s chiama q:

- stesso liv. di nesting
- Copiare gli elementi di v[]



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;  
...  
...  
...
```

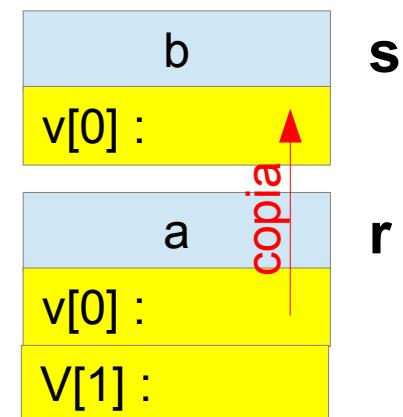


Chiamata a procedure definite esternamente:

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

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



Proprietà di questa implementazione

- Accesso alle variabili in tempo costante
 - Accesso a vettore + somma del puntatore ivi contenuto e dell'offset
 - Indipendente dai livelli 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 naïve 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

Linguaggi di Programmazione I – Lezione 6

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

8 aprile 2008

Analisi di oggetti e classi	3
Introduzione	4
Astrazioni chiave	5
Esempi	6
Oggetti e classi	7
Classi e oggetti UML	8
Attributi e metodi	9
Esercizio/Progetto	10
Relazioni tra classi	11
Introduzione	12
Ereditarietà	13
Generalizzazione	14
Specializzazione	15
Polimorfismo	16
Classi astratte	17
Assoc. e molteplicità	18
Associazioni compl.	19
Classe di associaz.	20
Assoc. qualificate	21
Rouli nelle ass.	22
Assoc. riflessive	23
Aggregazioni	24
Composizioni	25
Esercizio/Progetto	26
Analisi della dinamica del modello	27
Riepilogo	28
Modellazione dinamica	29
Responsabilità	30
Evoluzione del sistema	31
Diagrammi <i>Sequence</i> e <i>Collaboration</i>	32
Esercizio/Progetto	33
Diagrammi <i>State</i>	34
Esempio 1	35
Esempio 2	36

Esercizio/Progetto	37
Diagrammi <i>Activity</i>	38
Esempio 3	39
Esercizio/Progetto	40
Bibliografia	41
Bibliografia	41
Appendice A: Abusi	42
Aggregati	43
Gerarchie	44
Classi/Istanze 1	45
Classi/Istanze 2	46
Classi/Istanze 3	47
IsA 1	48
IsA 2	49
IsA 3	50
IsA 4	51
Appendice B: Codifica Java di relazioni	52
Generalità	53
Composizione	54
Aggregazione	55
Associazione	56

Panoramica della lezione

Analisi di oggetti e classi

Relazioni tra classi

Analisi della dinamica del modello

Bibliografia

LP1 – Lezione 6

2 / 56

Analisi di oggetti e classi

3 / 56

Introduzione

- La fase di analisi identifica gli oggetti richiesti in esecuzione per assicurare la funzionalità del sistema.
- Segue la fase di *Individuazione delle Specifiche* e degli *Use Case*, precede la fase del *Progetto del Sistema*.
- Definisce cosa deve fare il sistema.
- Evita di descrivere dettagli di progettazione e realizzazione.
- Pone l'accento sui componenti del sistema.

Durante questa fase bisognerebbe rispondere a domande:

- Quali sono gli oggetti del sistema?
- Quali sono le possibili classi?
- Come sono correlati gli oggetti?
- Quali sono le responsabilità di ciascun oggetto o classe?
- Come sono correlati oggetti e classi?

LP1 – Lezione 6

4 / 56

Astrazioni chiave

- Riferiscono ad una sottolista di parole all'interno della lista dei possibili oggetti.
- Rappresentano gli oggetti primari o principali nel sistema.
- Sono identificate dopo aver studiato ciascun possibile oggetto e aver deciso se è abbastanza importante per essere una Astrazione Chiave.

LP1 – Lezione 6

5 / 56

Esempi

Esempio 1

Siano stati individuati i seguenti oggetti possibili, in riferimento ad un sistema-tavolo:

- Tavolo
- Gamba
- Piano
- Colore
- Lunghezza
- Larghezza
- Altezza
- Peso
- Data di acquisto
- Materiale

Quali di questi sono componenti "veri" di un tavolo, e pertanto gli oggetti

all'interno di un sistema-tavolo?

Quali di questi sono attributi di altri degli oggetti?

La risposta è soggettiva e dipende da ciò che inizialmente era richiesto al modello. Ma una possibile lista di astrazioni chieva potrebbe essere:

- Tavolo
- Gamba
- Piano

Gli altri elementi, come il Peso o il Colore, sono attributi di uno (o più) oggetti.

Esempio 2

Siano stati individuati i seguenti oggetti possibili, in riferimento al sistema

Jolly-Hotel.

- Camera
- Hotel
- Ospite
- Prezzo Base
- Correzione Stagionale al Prezzo
- Prenotazione
- Receptionist

Ma una possibile lista di astrazioni chiave potrebbe essere:

- Camera
- Hotel
- Ospite
- Prenotazione

LP1 – Lezione 6

6 / 56

Oggetti e classi

Il passo successivo è quello di rappresentare gli aspetti logici e fisici del modello statico. I diagrammi utilizzati sono i Class Diagram e gli Object Diagram.

- I diagrammi Class mostrano le classi che dovrebbero costituire il sistema, insieme con tutte le possibili relazioni tra esse.
- I diagrammi Object rappresentano gli oggetti presenti nel sistema ad un certo istante di tempo.

Entrambi i diagrammi utilizzano la stessa sintassi di base e possono essere prodotti contemporaneamente.

Di solito i diagrammi Class sono prodotti prima di quelli Object (pensare prima in modo astratto, poi specificare e verificare), anche se alcuni analisti preferiscono lavorare in modo inverso (prima elencare i particolari, poi produrre le astrazioni).

LP1 – Lezione 6

7 / 56

Classi e oggetti UML

In aggiunta a quanto detto nelle precedenti lezioni:



- Gli oggetti non hanno la sezione dei metodi, poiché essi possiedono tutti i metodi della classe di cui sono istanze.
- Analogamente gli oggetti non hanno la sezione degli attributi, ma possono avere una sezione in cui sono elencati solo i *valori* degli attributi della classe che rendono unico quell'oggetto.

LP1 – Lezione 6

8 / 56

Attributi e metodi

- Non sono noti prima della fase di Progettazione.
- Alcuni, però, sono già ovvi e possono essere aggiunti anche nella fase di Analisi.
- In questo momento il fatto che un attributo o un metodo esista è già sufficiente.
- Non è essenziale aggiungere informazioni sul tipo di un attributo o sui parametri di un metodo.
- Gli attributi che si possono qui identificare sono i nomi del dizionario dati che non erano stati classificati come astrazioni chiave. Il fatto che essi siano stati utilizzati per descrivere il sistema, significa che sono rilevanti in qualche modo.

LP1 – Lezione 6

9 / 56

Esercizio/Progetto (Quarta parte)

1. Nel sistema scelto, identificare la lista delle astrazioni chiave dalla lista degli oggetti candidati.
2. Studiare le astrazioni chiave aggiungendo gli attributi e metodi che si possono già identificare.
3. Tracciare un diagramma Class usando le astrazioni chiave dei punti precedenti.

LP1 – Lezione 6

10 / 56

Relazioni tra classi

11 / 56

Introduzione

- C'è una classe che ha bisogno di un'altra per una certa funzionalità?
- Vi sono relazioni così forti che un oggetto non può esistere senza un altro?
- Vi sono classi che hanno strutture o attributi simili?
- Vi sono classi che esibiscono comportamenti simili?
- Vi sono classi che hanno una genealogia simile nell'ambito del dominio del problema?

Le relazioni tra classi che costruiremo rispondendo alle domande precedenti sono:

- | | |
|--|--|
| <ul style="list-style-type: none">■ Ereditarietà<ul style="list-style-type: none">◆ Generalizzazioni◆ Specializzazioni■ Polimorfismo■ Classi astratte | <ul style="list-style-type: none">■ Associazioni<ul style="list-style-type: none">◆ Aggregazione◆ Composizione◆ Associazioni complesse |
|--|--|

LP1 – Lezione 6

12 / 56

Ereditarietà

- Il concetto di ereditarietà descrive come si possano condividere attributi e funzionalità tra classi di natura o scopo simile.
- Si rivedano le lezioni precedenti per la notazione UML usata per le ereditarietà e per i concetti di:
 - classificazione*;
 - ereditarietà singola o multipla*;
 - sottoclassi o superclassi*.
- Vi sono due modi con cui si può aggiungere il concetto di ereditarietà al modello del sistema:
 - Generalizzazione.
 - Specializzazione.

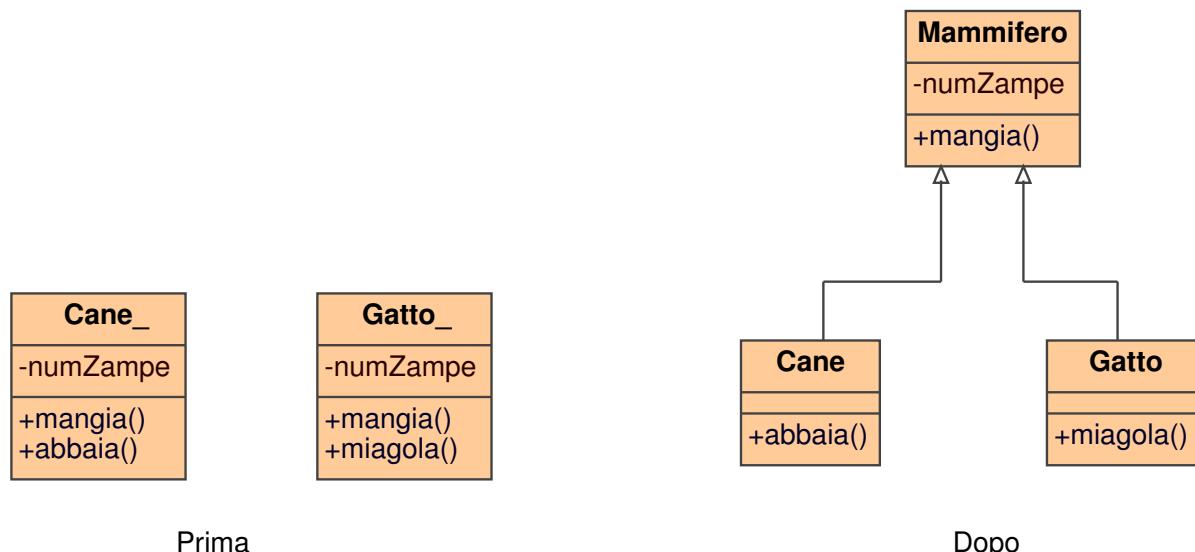
LP1 – Lezione 6

13 / 56

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 e semplifica le classi precedenti come estensioni della nuova (generalizzata).



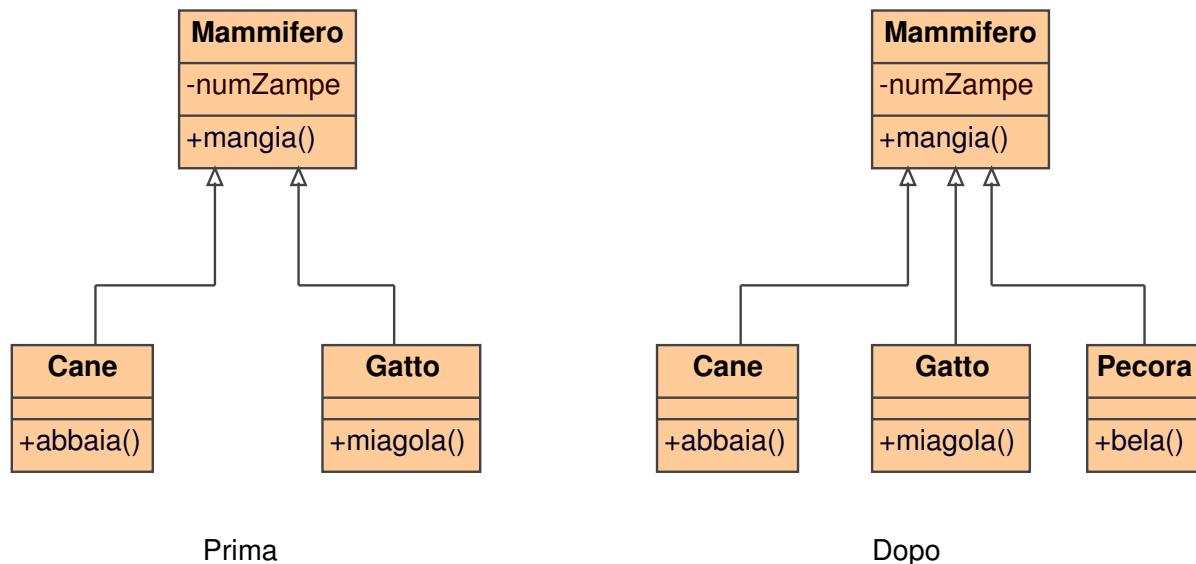
LP1 – Lezione 6

14 / 56

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.



LP1 – Lezione 6

15 / 56

Polimorfismo

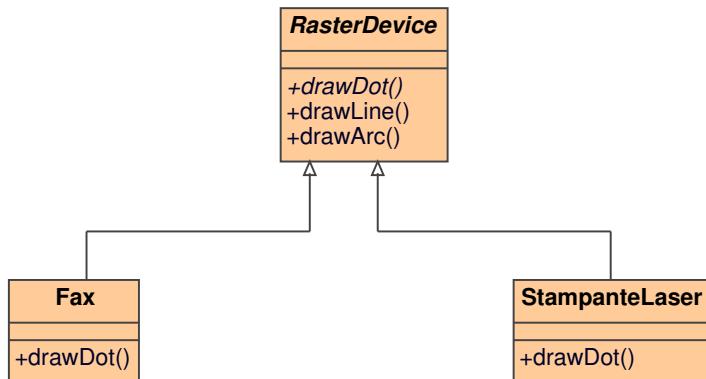
- Strettamente correlato con l'ereditarietà.
- Riguardando gli esempi precedenti si può dire che vi sono *molte forme* di Mammiferi nel sistema.
- L'effetto pratico è che un riferimento ad un Mammifero può essere usato sia per riferirsi ad un oggetto Cane, sia ad un oggetto Gatto.
- Senza polimorfismo sarebbe difficile scrivere metodi che possono operare su diversi tipi di oggetti.
- In una classe si può dichiarare una variabile per riferirsi ad un Mammifero prima che si sappia esattamente quante classi estendono Mammifero e quali saranno istanziate dal sistema in una particolare circostanza.

LP1 – Lezione 6

16 / 56

Classi astratte

- Contengono funzionalità incomplete (metodi astratti, privi di realizzazione) e non può 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 non realizzi tutti i metodi astratti ereditati.
- In UML i nomi di classi o metodi astratti sono in carattere *italico*.

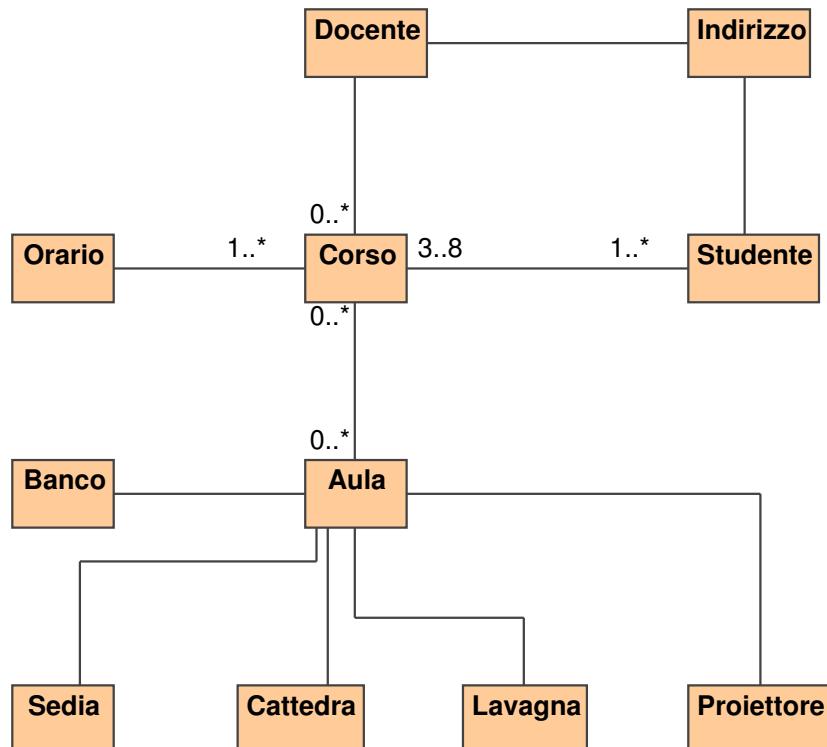


LP1 – Lezione 6

17 / 56

Associazioni e molteplicità

Rivedere le definizioni date nelle lezioni precedenti ed interpretare il Class Diagram:



LP1 – Lezione 6

18 / 56

Associazioni complesse

- Sono quelle in cui vi sono molteplicità maggiori di 1 in entrambi i ruoli.

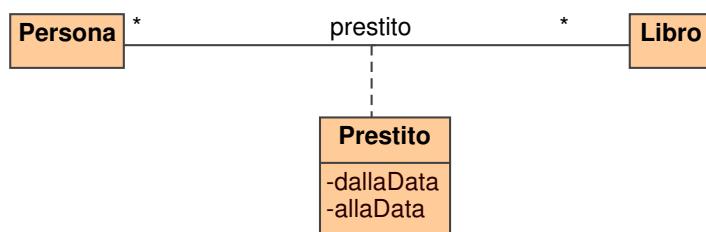


- È difficile pensare ad una possibile realizzazione.
- Due tecniche per la risoluzione:
 - ◆ Classe di associazioni.
 - ◆ Associazione qualificata.

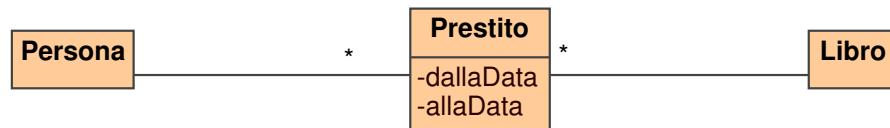
LP1 – Lezione 6

19 / 56

Classe di associazioni



- Il nome della classe deve essere lo stesso della associazione.
- Una Cl. di Ass. è usata per documentare la risoluzione di una associazione complessa, non fa parte delle astrazioni chiave.
- Restano nei diagrammi Class finché, nella fase di progettazione, non si decide di realizzarle.
- Per realizzarle, bisognará inserirle in altro modo nei diagrammi:

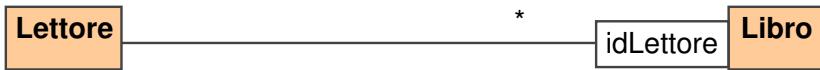


LP1 – Lezione 6

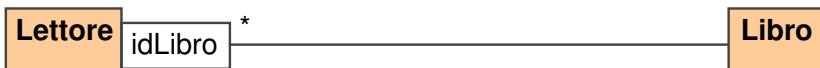
20 / 56

Associazioni qualificate

- Usate quando si vuole evidenziare che la realizzazione futura sarà mediante un array, una hash table, un dizionario ...
- In questo caso, se, per esempio, nella biblioteca ogni lettore (istanza di Lettore) ha il proprio unico codice identificativo (indice nell'array), ogni accesso dal punto di vista del Libro avviene mediante tale codice identificativo:



Viceversa, se nella biblioteca ogni libro è registrato usando un unico codice, il riferimento ad esso, dal punto di vista del Lettore, avviene mediante tale codice:



- La differenza realizzativa è che, mentre con una vera e propria classe di associazioni, le informazioni sul singolo prestito sono localizzate in una istanza di quella classe, nel caso delle associazioni qualificate, le informazioni sui prestiti sono delocalizzate e modificare un prestito significa modificare una istanza di Libro ed una istanza di Lettore.

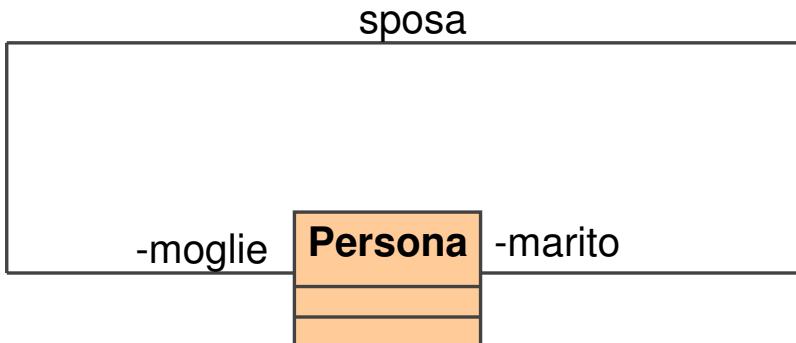
Ruoli nelle associazioni

- 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.



Assoc. 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.



LP1 – Lezione 6

23 / 56

Aggregazioni

- Sono forme di associazioni.
- Maggiore enfasi su come i due oggetti sono correlati nel sistema.
- Caratterizzata dalla relazione "ha un".
- Possono avere molteplicità.
- In UML:

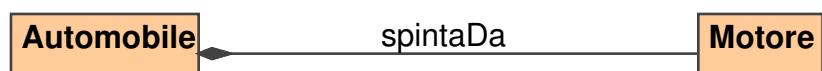


LP1 – Lezione 6

24 / 56

Composizioni

- Sono forme di associazioni.
- Maggiore enfasi su come i due oggetti sono correlati nel sistema.
- Caratterizzata dalla relazione "contiene sempre".
- Possono avere molteplicità.
- In UML:



LP1 – Lezione 6

25 / 56

Esercizio/Progetto (Quinta parte)

1. Usando le astrazioni chiave preparate nell'esercizio precedente, aggiungere linee di associazione tra classi.
2. Nominare le associazioni aggiunte al diagramma di Classe, aggiungendo indicatori di direzione se necessario.
3. Aggiungere valori di molteplicità alle associazioni.
4. Simulando l'esecuzione del sistema, tracciare un diagramma Object relativo ad una parte del diagramma Class ad un certo istante di tempo, in modo da verificare i valori di molteplicità.
5. Determinare e marcare le associazioni complesse nel diagramma Classe sviluppato negli esercizi precedenti.
6. Decidere quali attributi sono richiesti per risolvere ogni associazione complessa.
7. Risolvere le associazioni complesse nel modo più appropriato.
8. Riconsiderare il diagramma Class del proprio problema e ricercare le classi simili che possono essere generalizzate.
9. Aggiungere la classe generalizzata al diagramma e tracciare archi di ereditarietà tra le opportune classi.
10. Riconsiderare le specifiche del proprio problema. Può esistere una nuova classe che era sfuggita al diagramma Class? Se è così, aggiungere la classe e tracciare gli opportuni archi di ereditarietà.
11. Riprendere il diagramma Class e cercare, se possibile, di evidenziare le classi ed i metodi astratti.
12. Specificare quali debbano essere, nella gerarchia di ereditarietà, le classi che realizzano i metodi astratti.
13. Aggiungere i nomi di ruolo ad ogni associazione nel sistema in esame.
14. Evidenziare, se possibile, associazioni riflesive.
15. Individuare se possibile, nel problema in esame, le relazioni di aggregazione e composizione. Rappresentarle in modo appropriato, usando la sintassi UML.

LP1 – Lezione 6

26 / 56

Analisi della dinamica del modello

27 / 56

Riepilogo

- Formalizzazione del problema.
- Dizionario.
- Diagrammi Use Case.
- Scenari di uno Use Case.
- Astrazioni chiave.
- Diagrammi Class e Object.

LP1 – Lezione 6

28 / 56

Modellazione dinamica

Consiste nello specificare come gli oggetti operano e cooperano nel tempo.

Essa avviene due volte:

- durante la fase di analisi iniziale (quando si vuole essere sicuri che le operazioni siano possibili nel sistema);
- durante la fase di progetto fisico (per assegnare le descrizioni dei metodi alle classi appropriate).

In UML quattro diagrammi base:

- diagrammi Sequence;
- diagrammi Collaboration;
- diagrammi State;
- diagrammi Activity.

LP1 – Lezione 6

29 / 56

Responsabilità

Nel paradigma OO, "Responsabilità" significa:

- qualcosa che la classe conosce (stato);
- qualcosa che la classe sa (comportamento);
- qualcosa che l'oggetto conosce (stato);
- qualcosa che l'oggetto sa (comportamento).

Un comportamento definito in una interfaccia può essere realizzato in più di una classe. In questo caso si ha una realizzazione *polimorfica* di una responsabilità.

Il comportamento di una classe può essere distribuito in più sottoclassi oppure realizzato tutto nella stessa classe.

La qualità complessiva di progettazione OO è determinata da come e dove vengono distribuite le responsabilità.

Questo è un compito cruciale nella programmazione OO, tanto che sono stati sviluppati alcuni principi generali da usare come linee guida per l'assegnazione delle responsabilità (*Patterns*).

LP1 – Lezione 6

30 / 56

Evoluzione del sistema

Descrivere come cambia nel tempo un sistema significa considerare vari aspetti:

1. determinare se ciascuna operazione è una richiesta singola da parte di un utente o di altri sistemi;
2. determinare gli oggetti coinvolti in ogni singola operazione, come essi interagiscono e come ciascuna operazione può alterare lo stato interno degli oggetti.

I diagrammi usati in questo caso sono i *Sequence*, *Collaboration*, *State*.

LP1 – Lezione 6

31 / 56

Diagrammi Sequence e Collaboration

- Sono usati per descrivere gli scenari di uno stesso Use Case (si veda la lezione 5).
- Durante la fase di Analisi devono riflettere solo le interazioni.
- Durante la fase di Progettazione, ciascuna interazione sarà convertita nell'invocazione di un metodo nel linguaggio OO scelto.
- Per un certo Use Case dovrebbero essere prodotti tanti diagrammi Sequence e/o Collaboration, uno per ogni scenario di quello Use Case.
- Fare riferimento alla bibliografia e alla discussione della lezione 5 per la sintassi di diagrammi Sequence e Collaboration.

LP1 – Lezione 6

32 / 56

Esercizio/Progetto (Sesta parte)

1. Rivedere ed eventualmente correggere gli scenari per il sistema in esame.
2. Alla luce dei diagrammi Class e Object già sviluppati, tracciare i diagrammi Sequence per gli scenari utilizzati.
3. Tracciare in alternativa o a supplemento i diagrammi Collaboration.

LP1 – Lezione 6

33 / 56

Diagrammi State

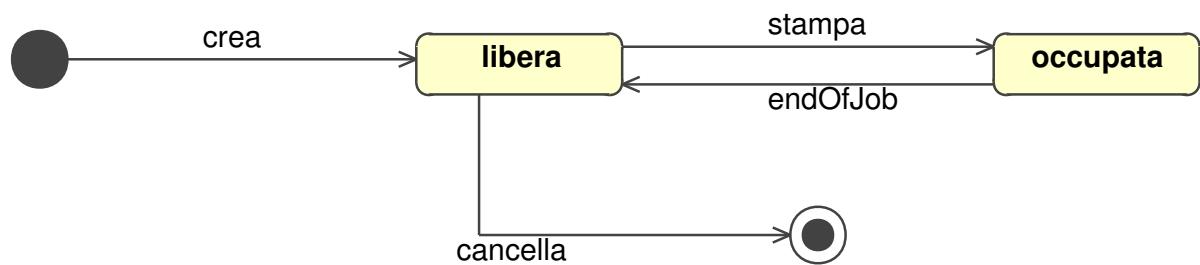
- Mostrano come cambia un oggetto nel tempo a causa dell'invocazione di metodi.
- Consistono di:
 - ◆ *Stati*: uno stato è l'insieme di valori degli attributi;
 - ◆ *Eventi*: un evento è lo stimolo che in un oggetto causa la transizione da uno stato all'altro.
- Sono importanti per descrivere gli stati legalmente assumibili dagli oggetti e la natura delle modifiche che possono legittimamente accadere nel tempo.
- Fare riferimento alla bibliografia e alla discussione della lezione 5 per la sintassi di diagrammi State.

LP1 – Lezione 6

34 / 56

Esempio 1

Diagramma di una istanza di Stampante:

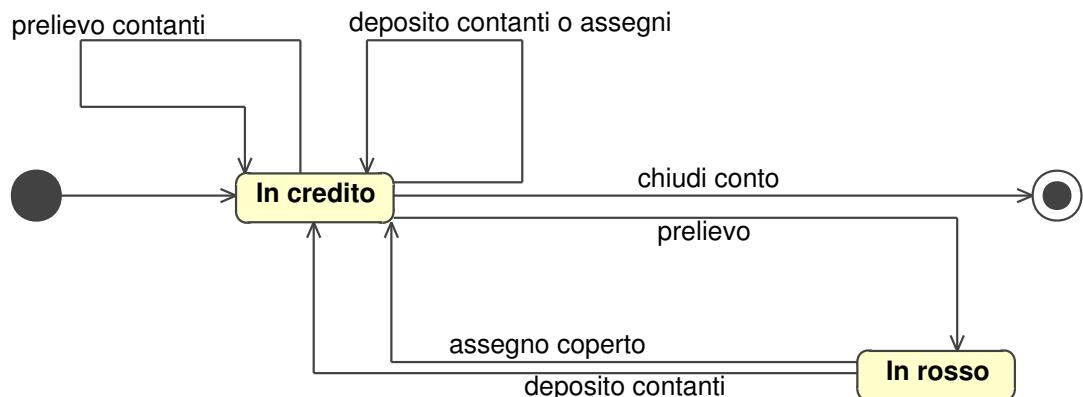


LP1 – Lezione 6

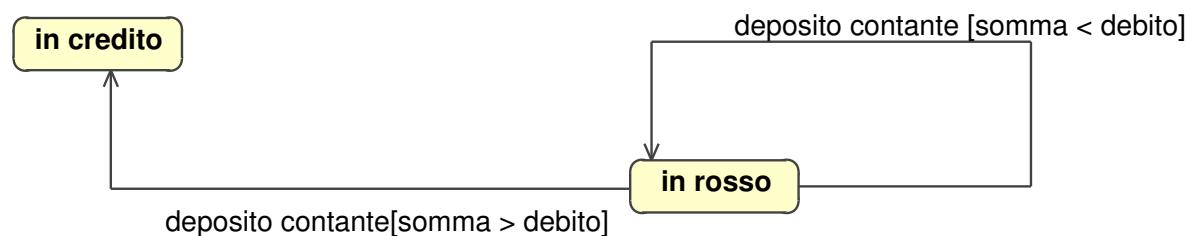
35 / 56

Esempio 2

Diagramma di una istanza di ContoBanca:



Sono possibili anche le transizioni con “guardie”:



LP1 – Lezione 6

36 / 56

Esercizio/Progetto (Settima parte)

1. Rappresentare il diagramma State per alcune classi rappresentative del progetto in esame.

LP1 – Lezione 6

37 / 56

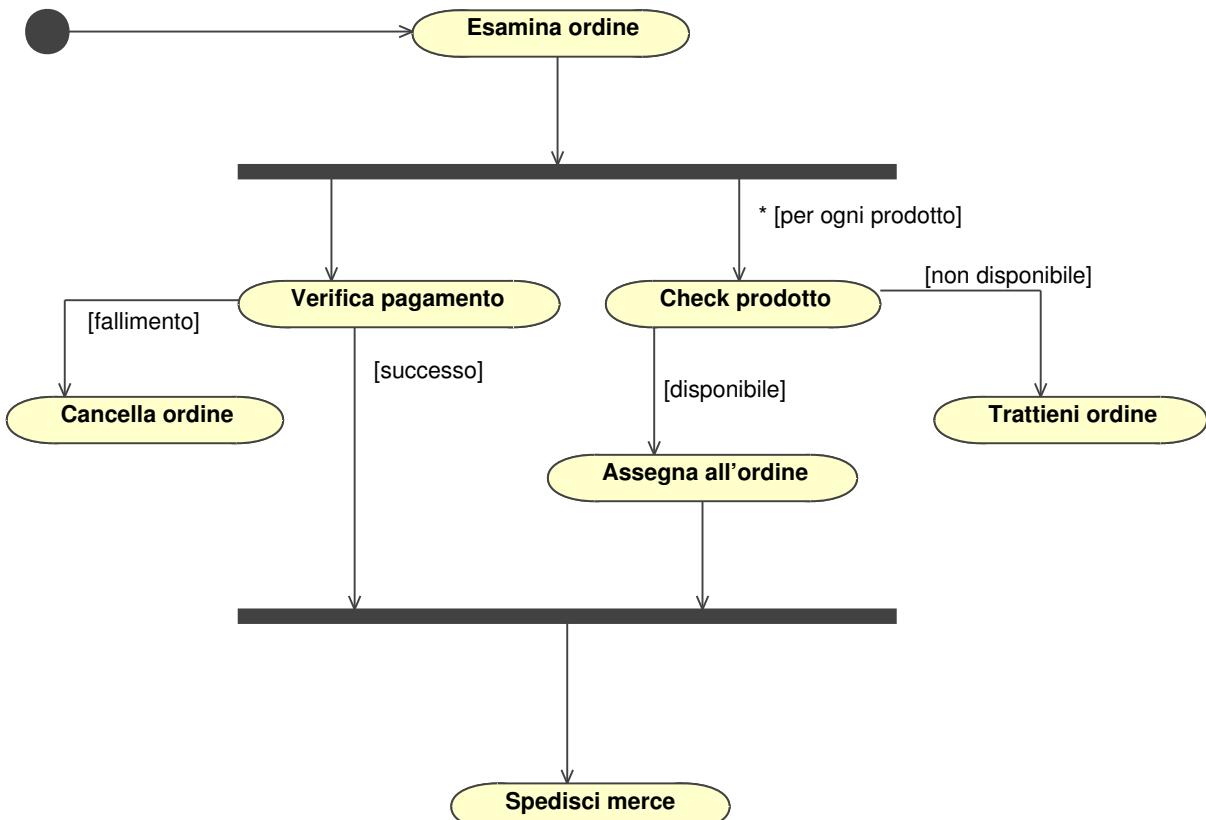
Diagrammi Activity

- Mostrano il diagramma di flusso delle attività.
- Un solo diagramma per ciascuno Use Case che riassume tutti i diagrammi Sequence.
- Fare riferimento alla bibliografia e alla discussione della lezione 5 per la sintassi di diagrammi Activity.

LP1 – Lezione 6

38 / 56

Esempio 3



LP1 – Lezione 6

39 / 56

Esercizio/Progetto (Ottava parte)

1. Rappresentare il diagramma Activity relativo ad alcuni Use Case rappresentativi del progetto in esame.

LP1 – Lezione 6

40 / 56

Bibliografia

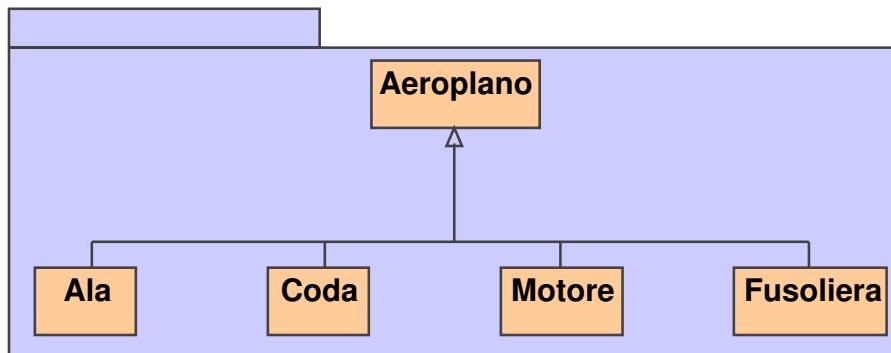
41 / 56

Bibliografia

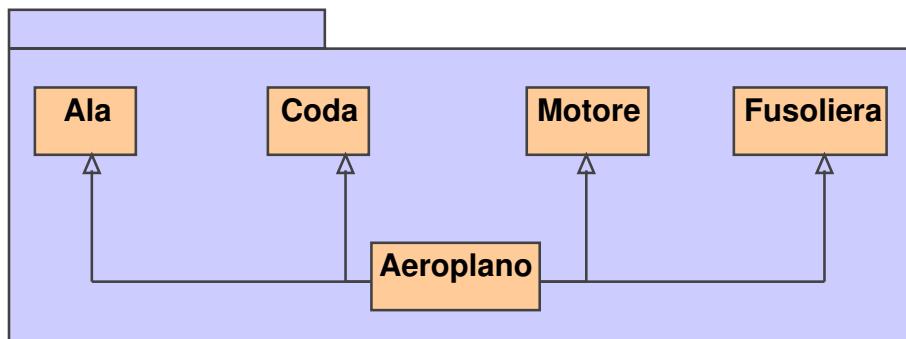
- Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language Reference Manual*. Addison-Wesley.
- Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*. Addison-Wesley.
- Fowler M. *UML distilled*. Addison-Wesley.

LP1 – Lezione 6

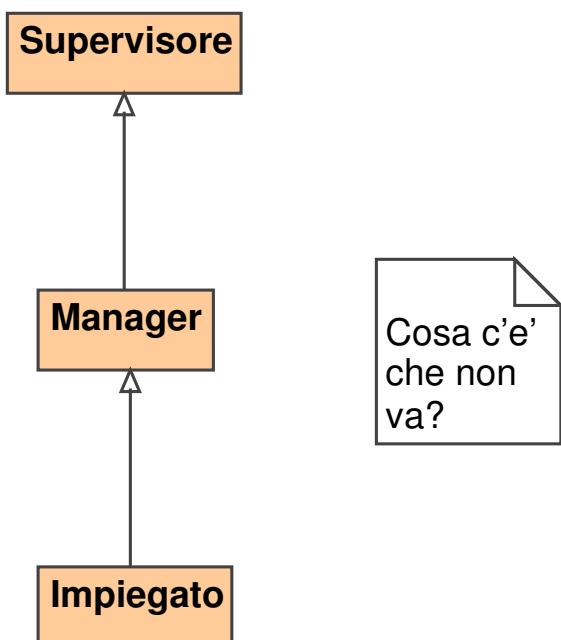
41 / 56

Aggregati

Cosa c'e'
che non
va?



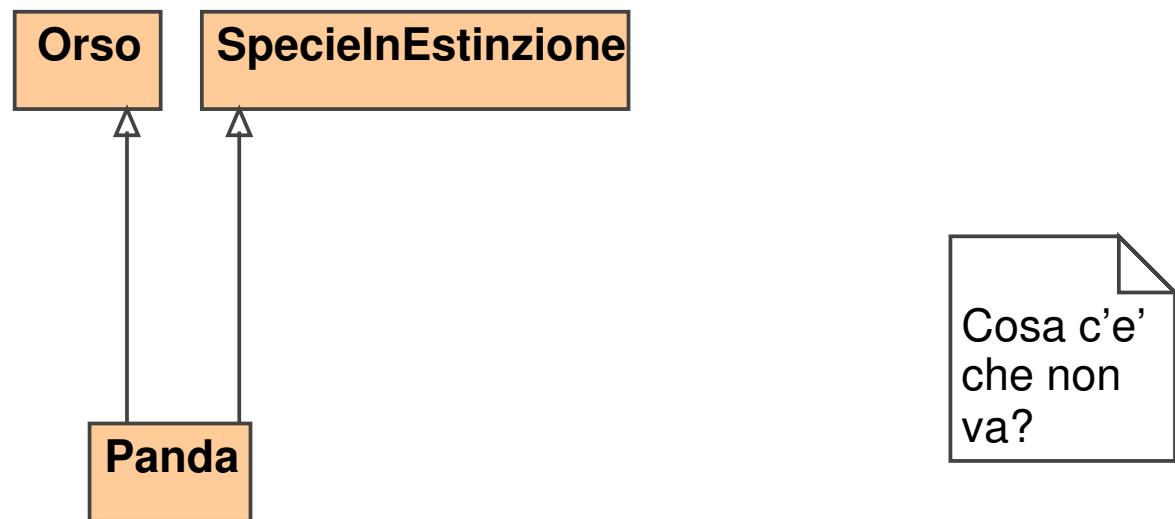
Gerarchie



LP1 – Lezione 6

44 / 56

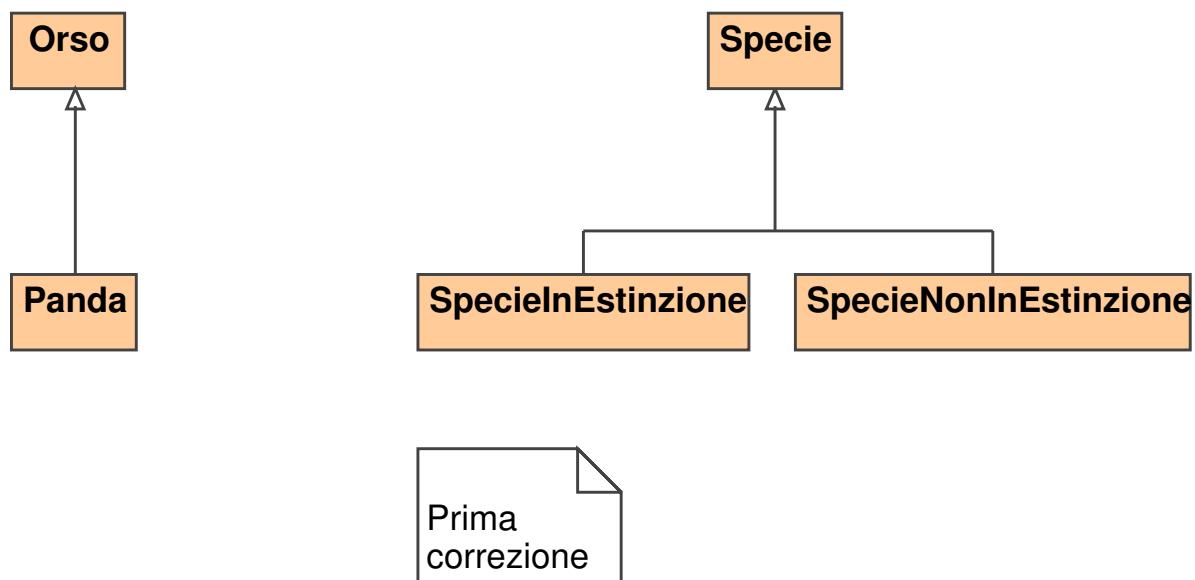
Classi/Istanze 1



LP1 – Lezione 6

45 / 56

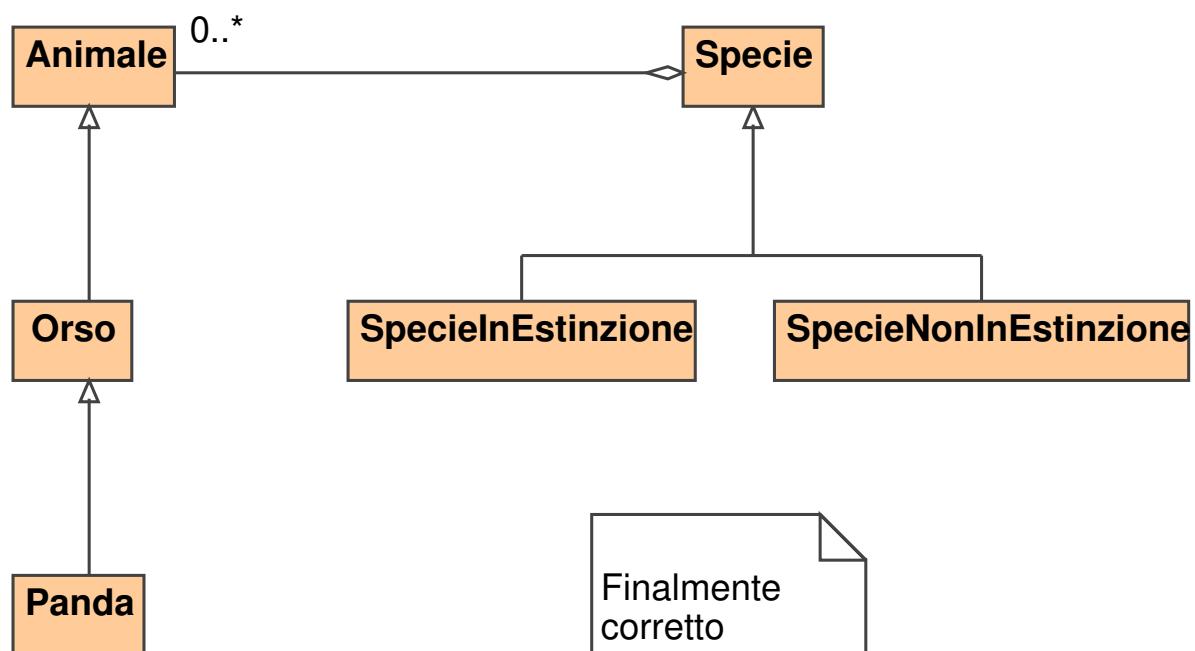
Classi/Istanze 2



LP1 – Lezione 6

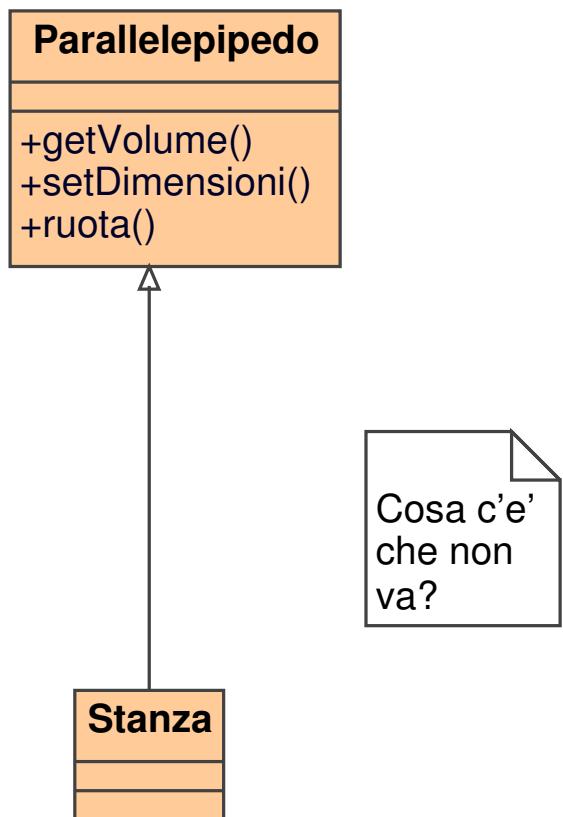
46 / 56

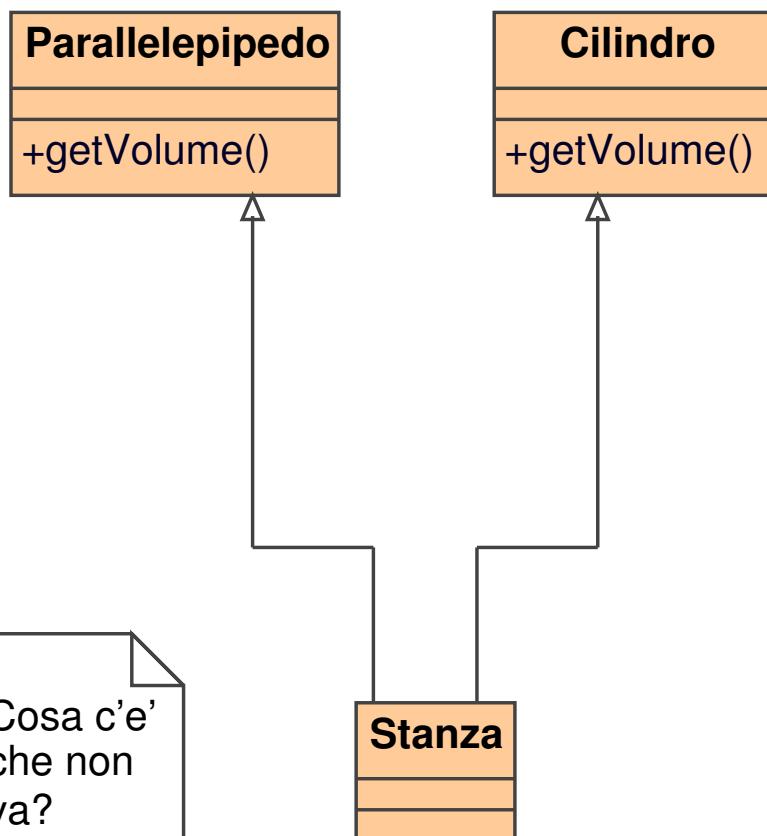
Classi/Istanze 3



LP1 – Lezione 6

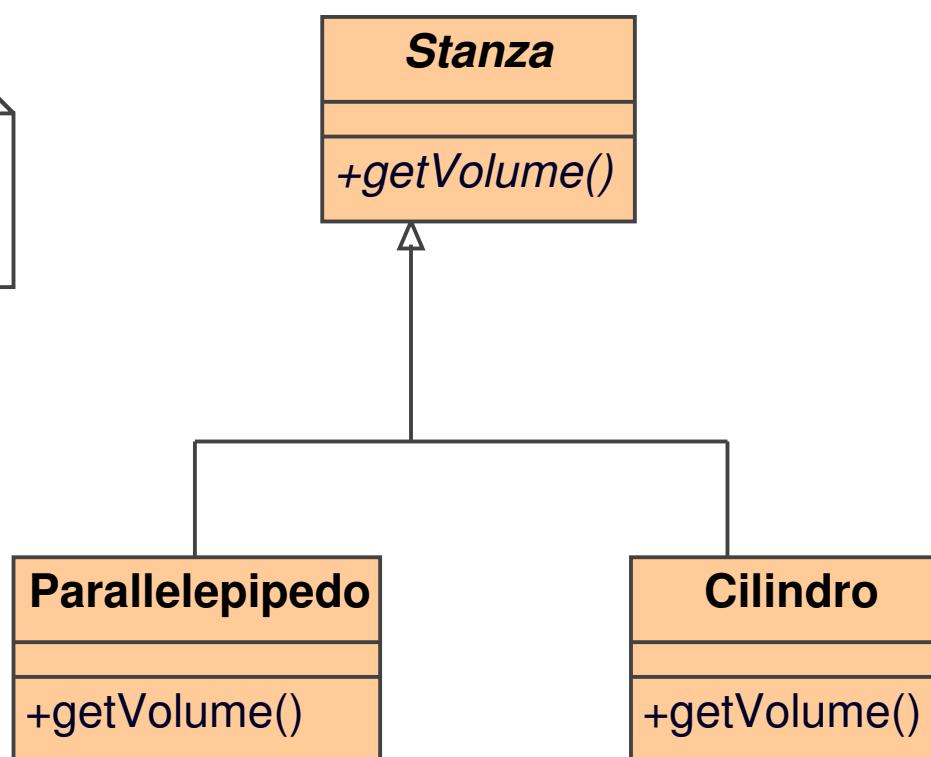
47 / 56





IsA 3

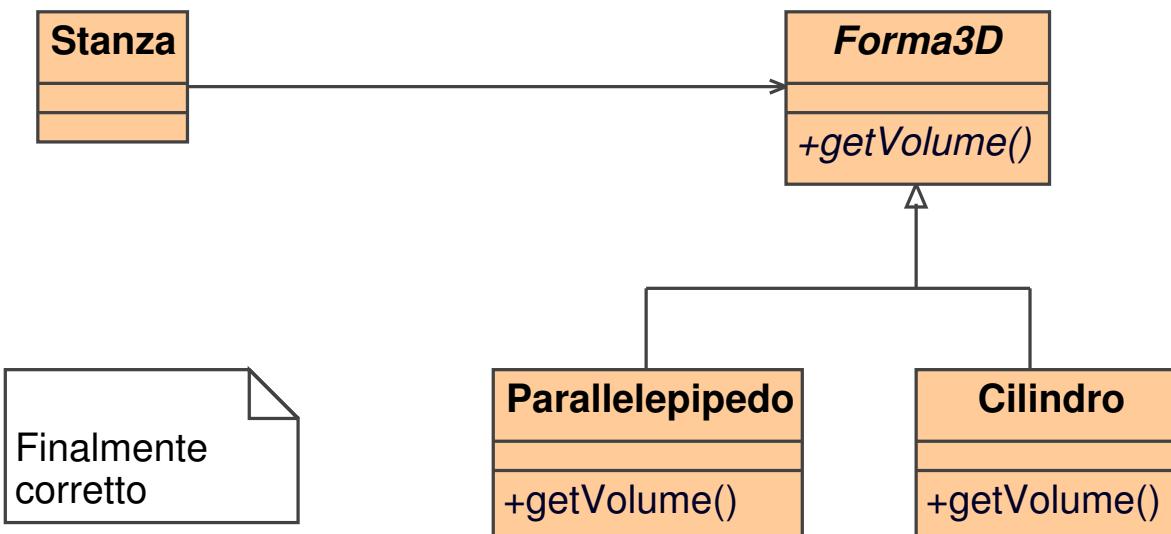
Cosa c'e'
che non
va?



LP1 – Lezione 6

50 / 56

IsA 4



LP1 – Lezione 6

51 / 56

Generalità

Una composizione è anche una aggregazione, una aggregazione è anche una associazione. Pertanto:

- usare associazioni è sempre corretto;
- bisogna fare attenzione ad usare le altre relazioni;
- in una aggregazione (composizione), una operazione eseguita sull'aggregato (composto) viene propagata su uno o più aggreganti (componenti);
- il tutto deve sapere delle parti, ma non viceversa;
- la composizione 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;
- nella aggregazione le parti continuano ad esistere anche dopo la morte del tutto;
- nella semplice associazione le vite degli oggetti, sia pure correlate, esistono in modo totalmente indipendente.

LP1 – Lezione 6

53 / 56

Composizione

```
class Automobile
{
    Motore motore;
    // altri attributi

    Automobile (int cilindrata, int numeroCilindri,
                String marca, String modello)
    {
        motore = new Motore (cilindrata, numeroCilindri);
        // altro codice
    }

    // altri metodi
}
```

LP1 – Lezione 6

54 / 56

Aggregazione

```
class Automobile
{
    Motore motore;
    // altri attributi

    Automobile (Motore motore, String marca, String modello)
    {
        this.motore = motore;
        // altro codice
    }

    // altri metodi
}

// altrove nel codice
Motore m = new Motore (1400, 3);
Automobile a = new Automobile (m, "Seat", "Ibiza");
```

LP1 – Lezione 6

55 / 56

Associazione

```
class Automobile
{
    Persona proprietario;
    // altri attributi

    setProprietario (Persona proprietario)
    {
        this.proprietario = proprietario;
    }

    // altri metodi
}

// altrove nel codice
Automobile a = new Automobile ("Seat", "Ibiza");
Persona io = new Persona ("Marcello", "Sette");
a.setProprietario(io);
```

LP1 – Lezione 6

56 / 56

Linguaggi di Programmazione I – Lezione 10

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
<http://sette.dnsalias.org>

8 maggio 2008

Array	3
Dichiarazione	4
Creazione (1)	5
Creazione (2)	6
Inizializzazione (1)	7
Inizializzazione (2)	8
Esempio	9
Multidimensioni	10
Estremi	11
Assegnazione	12
Ridimensionamento	13
Copia	14
Aiutare il GC	15
Esercizi	16
Esercizi	16
Questionario	17
D 1	18
D 2	19
D 3	20
D 4	21
D 5	22
D 6	23
D 7	24
D 8	25
D 9	26
D 10	27
D 11	28
D 12	29
D 13	30
D 14	31
D 15	32
D 16	33
D 17	34
D 18	35

D 19	36
D 20	37
D 21	38
D 22	39
D 23	40

Array

3 / 40

Dichiarazione

- Si possono dichiarare array di tipi primitivi o di riferimenti ad oggetti:

```
char s[];  
Point p[];  
  
char [] s;  
Point [] p;
```

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

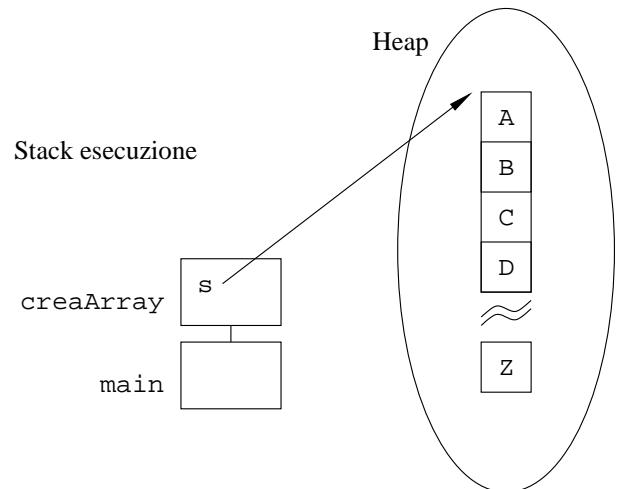
- NON viene qui generato l'oggetto: la generazione richiede ancora l'uso di `new`.
- Nella dichiarazione NON si deve specificare la dimensione dell'array.
- Se si dichiarano più array in uno stesso enunciato usando le parentesi quadre a sinistra, le parentesi sono applicate a tutte le variabili alla loro destra.
- Esempio:

```
int a[], b; // a e' riferimento ad array, b e' int  
  
int [] a, b; // a e b entrambi riferimenti ad array
```

Creazione (1)

- Per creare l'array si deve usare new.
- Esempio:

```
public char[] creaArray() {  
    char[] s;  
  
    s = new char[26];  
    for (int i=0; i<26; i++) {  
        s[i] = (char) ('A' + i);  
    }  
  
    return s;  
}
```



`s = new char[26]`; crea un array di 26 caratteri: essi sono inizializzati al loro valore di default ('\u0000' per char). I valori sono accessibili nel range da 0 a 25: ogni tentativo di accesso oltre questo range causerà il lancio di una eccezione a runtime.

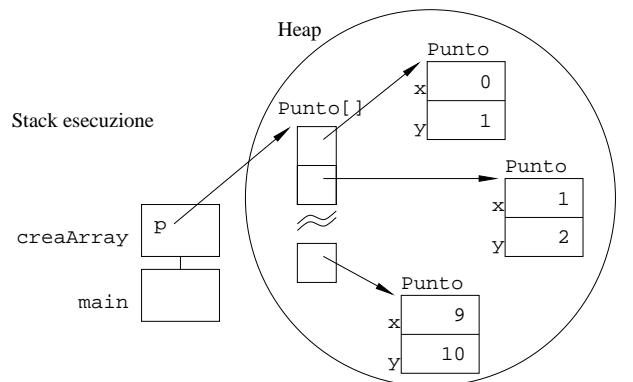
LP1 – Lezione 10

5 / 40

Creazione (2)

- Un altro esempio per array di oggetti:

```
public Punto[] creaArray() {  
    Punto[] p;  
  
    p = new Punto[10];  
    for ( int i=0; i<10; i++ ) {  
        p[i] = new Punto(i, i+1);  
    }  
  
    return p;  
}
```



`p = new Punto[10]`; anche questa volta crea un array di 10 **riferimenti** ad oggetti `Punto`: essi sono inizializzati al loro valore di default null.

Questa volta è cruciale la creazione dei singoli oggetti, prima del loro uso.

Nota: una sintassi alternativa, ma forse meno leggibile, per il metodo precedente poteva essere:

```
public Punto creaArray() []
```

LP1 – Lezione 10

6 / 40

Inizializzazione (1)

Poiché l'inizializzazione delle variabili è cruciale, Java fornisce due metodi abbreviati per gli array (oltre il metodo diretto che utilizza un loop). 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";
```

LP1 – Lezione 10

7 / 40

Inizializzazione (2)

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

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

ma, 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});  
    }  
}
```

Attenzione: NON si deve specificare la dimensione in una creazione di un array anonimo.

```
new Object[2] {null, new Object()}; // illegale
```

LP1 – Lezione 10

8 / 40

Esempio

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. Per esempio, per creare un riferimento ad un tipo primitivo:

```
public class PrimitiveReference {  
    public static void main(String args[]) {  
        int [] mioValore = { 1 };  
        modifica(mioValore);  
        System.out.println("mioValore contiene " +  
            mioValore[0]);  
    }  
    public static void modifica(int [] valore) {  
        valore[0]++;  
    }  
}
```

Multidimensioni

- Array di array:

```
int dueDim [][] = new int [3][];  
dueDim[0] = new int [5];  
dueDim[1] = new int [5];  
dueDim[2] = new int [5];  
  
int dueDim [][] = new int [] [3]; // illegale
```

- Array non rettangolare di array:

```
int dueDim [][] = new int [3][];  
dueDim[0] = new int [2];  
dueDim[1] = new int [5];  
dueDim[2] = new int [8];
```

- Array rettangolare di array:

```
int dueDim [][] = new int [4][5];
```

- Costruzione e inizializzazione:

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

Estremi

- Indice iniziale 0.
- Numero di elementi è parte dell'oggetto array, nell'attributo `length`.
- Esempio:

```
int list[] = new int [10];
for (int i=0; i<list.length; i++) {
    System.out.println(list[i]);
}
```

- Accesso oltre i limiti causa il lancio di una eccezione a runtime.
- Cosa stampa il brano seguente?

```
int m[][] = new int [10][5];
System.out.println("m.length vale      " + m.length);
System.out.println("m[0].length vale " + m[0].length);
```

Assegnazione

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

```
int [] vettore;
int [][] matrice = new int[3][];

vettore = matrice; // illegale

int [] v = new int[6];
vettore = v;        // OK
```

Ridimensionamento

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

```
int elements[] = new int [6];
elements = new int [10];
```

- Che ne è del vecchio array?

Copia

Il metodo `System.arraycopy()`:

```
// array originale  
int vecchi[] = {1, 2, 3, 4};  
  
// nuovo array più lungo  
int nuovo[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};  
  
// copia tutti i vecchi elementi nel nuovo array  
System.arraycopy(vecchi, 0, nuovo, 0, vecchi.length);
```

Attenzione: il 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.

LP1 – Lezione 10

14 / 40

Aiutare il GC

Il garbage collector non è onnipotente. Alcune volte ha bisogno di un piccolo “aiuto”. Per esempio, supponiamo di dover operare su una pila usando il metodo:

```
public Object pop() {  
    return pila[indice--];  
}
```

Se l'utilizzatore del metodo abbandona il valore di ritorno ricevuto, esso non sarà eleggibile per GC, finché il riferimento ad esso nell'array pila non sarà sovrascritto. Questo potrebbe richiedere tempi lunghissimi. Più correttamente:

```
public Object pop() {  
    Object valore = pila[indice];  
    pila[indice--] = null;  
    return valore;  
}
```

LP1 – Lezione 10

15 / 40

Esercizi

16 / 40

Esercizi

1. Scrivere un programma che stampi i valori degli argomenti della linea di comando.
2. Estendere l'esercizio della lezione precedente, usando gli array per realizzare la molteplicità nella relazione (composizione) tra una banca e i propri clienti.

LP1 – Lezione 10

16 / 40

D 1

Un tipo di dato con segno ha un ugual numero di valori positivi e negativi.

- A. Vero
- B. Falso

LP1 – Lezione 10

18 / 40

D 2

Scegliere gli identificatori legali tra questi:

- A. StringaLunghissimaSenzaSignificato
- B. \$int
- C. bytes
- D. \$1
- E. finals

LP1 – Lezione 10

19 / 40

D 3

Quali delle seguenti segnature sono valide per il metodo main?

- A. public static void main()
- B. public static void main(String arg[])
- C. public void main(String[] arg)
- D. public static void main(String[] args)
- E. public static int main(String[] arg)

LP1 – Lezione 10

20 / 40

D 4

Se in un file sorgente sono presenti tutti e tre gli elementi top-level, in quale ordine devono apparire?

- A. import, package, class.
- B. class, import, package.
- C. package per primo, l'ordine degli altri non importa.
- D. package, import, class.
- E. import per primo, l'ordine degli altri non importa.

LP1 – Lezione 10

21 / 40

D 5

Si consideri la seguente linea di codice:

```
int [] x = new int [25];
```

Dopo l'esecuzione, quali delle seguenti affermazioni sono vere?

- A. x[0] è 0.
- B. x è indefinito.
- C. x è 0.
- D. x[0] è null.
- E. x.length è 25.

LP1 – Lezione 10

22 / 40

D 6

Qual è l'output della seguente applicazione:

```
class D6 {  
    public static void main(String args[]) {  
        Scatola s = new Scatola();  
        s.interno = 100;  
        s.aumenta(s);  
        System.out.println(s.interno);  
    }  
}  
  
class Scatola {  
    public int interno;  
    public void aumenta(Scatola scatola) {  
        scatola.interno++;  
    }  
}
```

A. 0

B. 1

C. 100

D. 101

LP1 – Lezione 10

23 / 40

D 7

Qual è l'output della seguente applicazione:

```
class D7 {  
    public static void main(String args[]) {  
        double d = 12.3;  
        Decremento dec = new Decremento();  
        dec.decrementa(d);  
        System.out.println(d);  
    }  
}  
  
class Decremento {  
    public void decrementa(double dec) {  
        dec = dec - 1.0;  
    }  
}
```

LP1 – Lezione 10

24 / 40

D 8

Come si può forzare la garbage collection di un oggetto?

- A. Il garbage collector non può essere forzato.
- B. Con una chiamata a `System.gc()`.
- C. Con una chiamata a `System.gc()`, passando il riferimento all'oggetto.
- D. Con una chiamata a `Runtime.gc()`.
- E. Ponendo tutti i riferimenti a quell'oggetto a `null`.

LP1 – Lezione 10

25 / 40

D 9

Qual è il range di valori per una variabile di tipo `short`?

- A. Dipende dall'hardware che ospita la JVM.
- B. $0 \dots 2^{16} - 1$
- C. $0 \dots 2^{32} - 1$
- D. $-2^{15} \dots 2^{15} - 1$
- E. $-2^{31} \dots 2^{31} - 1$

LP1 – Lezione 10

26 / 40

D 10

Qual è il range di valori per una variabile di tipo byte?

- A. Dipende dall'hardware che ospita la JVM.
- B. $0 \dots 2^8 - 1$
- C. $0 \dots 2^{16} - 1$
- D. $-2^7 \dots 2^7 - 1$
- E. $-2^{15} \dots 2^{15} - 1$

LP1 – Lezione 10

27 / 40

D 11

Quali sono i valori di x, a, b dopo l'esecuzione del codice:

```
int x, a = 6, b = 7;  
x = a++ + b++;
```

- A. x=15, a=7, b=8
- B. x=15, a=6, b=7
- C. x=13, a=7, b=8
- D. x=13, a=6, b=7

LP1 – Lezione 10

28 / 40

D 12

Quali delle seguenti espressioni sono legali?

- A. int x=6; x=!x;
- B. int x=6; if (!(x>3)) {}
- C. int x=6; x=~x;

LP1 – Lezione 10

29 / 40

D 13

Quali delle seguenti espressioni risultano in un valore positivo in x?

- A. int x=-1; x = x >>> 5;
- B. int x=-1; x = x >>> 32;
- C. byte x=-1; x = x >>> 5;
- D. int x=-1; x = x >> 5;

LP1 – Lezione 10

30 / 40

D 14

Quali delle seguenti espressioni sono legali?

- A. String x="Ciao"; int y=7; x += y;
- B. String x="Ciao"; int y=7; if (x == y) {}
- C. String x="Ciao"; int y=7; x = x + y;
- D. String x="Ciao"; int y=7; y = y + x;
- E. String x=null;
int y = (x!=null) &&
(x.length()>0) ? x.length() : 0;

LP1 – Lezione 10

31 / 40

D 15

Qual è il risultato dell'esecuzione del seguente codice?

```
public class Xor {  
    public static void main(String args[]) {  
        byte b = 10; // 00001010 binario  
        byte c = 15; // 00001111 binario  
        b = (byte)(b ^ c);  
        System.out.println("b vale " + b);  
    }  
}
```

- A. b vale 10
- B. b vale 5
- C. b vale 250
- D. b vale 245

LP1 – Lezione 10

32 / 40

D 16

Qual è il risultato della compilazione ed esecuzione del seguente codice?

```
1. public class Condizionale {  
2.     public static void main(String args[]) {  
3.         int x = 4;  
4.         System.out.println("Il valore e' " +  
5.             ((x > 4) ? 99.99 : 9));  
6.     }  
7. }
```

- A. Il valore e' 99.99
- B. Il valore e' 9
- C. Il valore e' 9.0
- D. Un errore di compilazione alla linea 5

LP1 – Lezione 10

33 / 40

D 17

Qual è l'output del seguente frammento di codice?

```
int x=3; int y=-10;
System.out.println(y % x);
```

- A. 0
- B. 1
- C. -1
- D. -3

LP1 – Lezione 10

34 / 40

D 18

Qual è l'output del seguente frammento di codice?

```
int x=1;
String [] nomi = {"Mario", "Anna", "Carlo"};
nomi[--x] += ".";
for (int i=0; i < nomi.length; i++) {
    System.out.println(nomi[i]);
}
```

- A. L'output include Mario. con un punto finale.
- B. L'output include Anna. con un punto finale.
- C. L'output include Carlo. con un punto finale.
- D. Nessun nome stampato ha il punto finale.
- E. Viene lanciata l'eccezione `ArrayIndexOutOfBoundsException`.

LP1 – Lezione 10

35 / 40

D 19

Quali linee fanno parte dell'output del seguente codice?

```
for (int i=0; i<2; i++) {
    for (int j=0; j<3; j++) {
        if (i == j) {
            continue;
        }
        System.out.println("i=" + i + " j=" + j);
    }
}
```

- A. i=0 j=0
- B. i=0 j=1
- C. i=0 j=2
- D. i=1 j=0
- E. i=1 j=1
- F. i=1 j=2

LP1 – Lezione 10

36 / 40

D 20

Quali linee fanno parte dell'output del seguente codice?

```
esterno: for (int i=0; i<2; i++) {  
    for (int j=0; j<3; j++) {  
        if (i == j) {  
            continue esterno;  
        }  
        System.out.println("i=" + i + " j=" + j);  
    }  
}
```

- A. i=0 j=0
- B. i=0 j=1
- C. i=0 j=2
- D. i=1 j=0
- E. i=1 j=1
- F. i=1 j=2

D 21

Quali tra queste sono costruzioni legali di loop?

A.

```
while (int i<7) {  
    i++;  
    System.out.println("i=" + i);  
}
```

B.

```
int i=3;  
while (i) {  
    System.out.println("i=" + i);  
}
```

C.

```
int j=0;  
for (int k=0; j + k != 10;  
     j++, k++) {  
    System.out.println(  
        "j=" + j + " k=" + k);  
}
```

D.

```
int j=0;  
do {  
    System.out.println(  
        "j=" + j++);  
    if (j == 3) {continue loop;}  
} while (j < 10);
```

D 22

Qual è l'output di questo frammento di codice?

```
int x=0, y=4, z=5;
if ( x>2 ) {
    if ( y<5 ) {
        System.out.println("uno");
    } else {
        System.out.println("due");
    }
} else if ( z>5 ) {
    System.out.println("tre");
} else {
    System.out.println("quattro");
}
```

- A. uno
- B. due
- C. tre
- D. quattro

LP1 – Lezione 10

39 / 40

D 23

Dato il codice:

```
1. int j=2;
2. switch (j) {
3.     case 2:
4.         System.out.print("2");
5.     case 2+1:
6.         System.out.print("3");
7.         break;
8.     default:
9.         System.out.print(j);
10.    break;
11. }
12. System.out.println();
```

Quale dei seguenti enunciati è vero?

- A. Il codice è illegale a causa dell'espressione alla linea 5.
- B. I tipi accettabili per la variabile di controllo di uno switch sono byte, short, int, long.
- C. L'output è 2.
- D. L'output è 23.
- E. L'output è 232.

LP1 – Lezione 10

40 / 40

Linguaggi di Programmazione I – Lezione 13

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
<http://sette.dnsalias.org>

22 maggio 2008

Classi astratte	3
Intro (1)	4
Intro (2)	5
Intro (3)	6
Problema A	7
Soluzione A.1	8
Soluzione A.2	9
Soluzione A.3	10
Problema B	11
Soluzione B	12
Interfacce	13
Generalità	14
Sintassi	15
Esempio (1)	16
Esempio (2)	17
Esempio (3)	18
Esempio (4)	19
Esempio (5)	20
Esempio (6)	21
Esempio (7)	22
Esempio (8)	23
Vantaggi	24
Esercizi	25
Esercizi	25
Casting di riferimenti	26
Introduzione	27
AutoConversioni (1)	28
AutoConversioni (2)	29
Esempi (1)	30
Esempi (2)	31
Esempi (3)	32
Casting (1)	33
Casting (2)	34

Casting (3).....	35
Casting (4).....	36
Esempi	37
Questionario	38
D 1.....	39
D 2.....	40
D 3.....	41
D 4.....	42
D 5.....	43
D 6.....	44

Classi astratte e interfacce

Classi astratte

Interfacce

Esercizi

Casting di riferimenti

Questionario

LP1 – Lezione 13

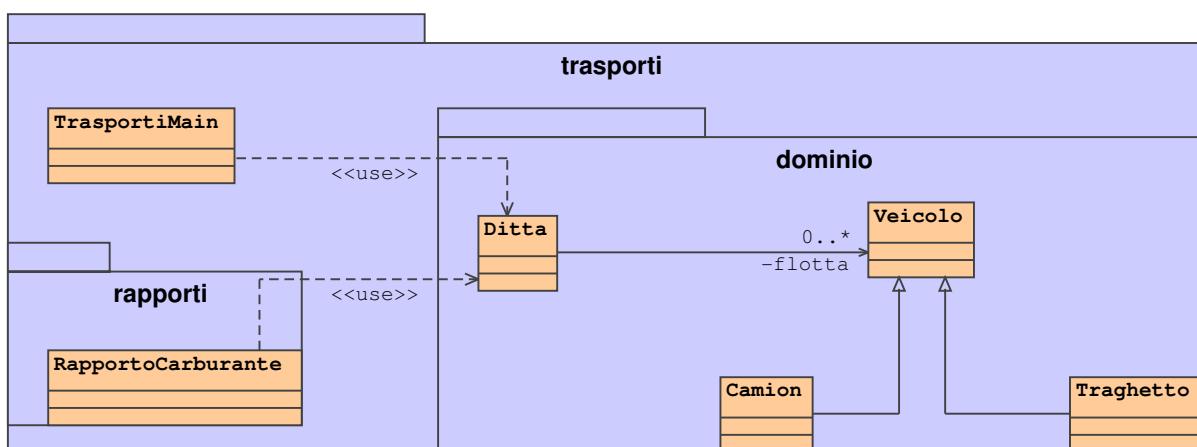
2 / 44

Classi astratte

3 / 44

Intro (1)

Supponiamo che si voglia un rapporto settimanale sul consumo dei veicoli di una ditta di trasporti.
Il diagramma UML potrebbe essere questo:



Il codice che realizza le due classi esterne al dominio sia:

LP1 – Lezione 13

4 / 44

Intro (2)

```
public class TrasportiMain {
    public static void main(String[] args) {
        Ditta d = Ditta.getDitta(); // Singoletto

        // popola la flotta di veicoli
        d.addVeicolo(new Camion(10000.0));
        d.addVeicolo(new Camion(15000.0));
        d.addVeicolo(new Traghetto(500000.0));
        d.addVeicolo(new Camion(9500.0));
        d.addVeicolo(new Traghetto(750000.0));

        RapportoCarburante rapporto = new RapportoCarburante();
        rapporto.generaTesto(System.out);
    }
}
```

LP1 – Lezione 13

5 / 44

Intro (3)

```
public class RapportoCarburante {  
    public void generaTesto(PrintStream output) {  
        Ditta d = Ditta.getDitta(); // Singoletto  
        Veicolo v;  
        double carburante;  
        double totale = 0.0;  
  
        for (int i=0; i<d.getDimFlotta(); i++) {  
            v = d.getVeicolo(i);  
            carburante = v.valutaConsumoPerKm() * v.valutaDistanza();  
            output.println("Consumo del veicolo " + v.getNome() +  
                           ": " + carburante);  
            totale += carburante;  
        }  
        output.println("Consumo totale: ", totale);  
    }  
}
```

LP1 – Lezione 13

6 / 44

Problema A

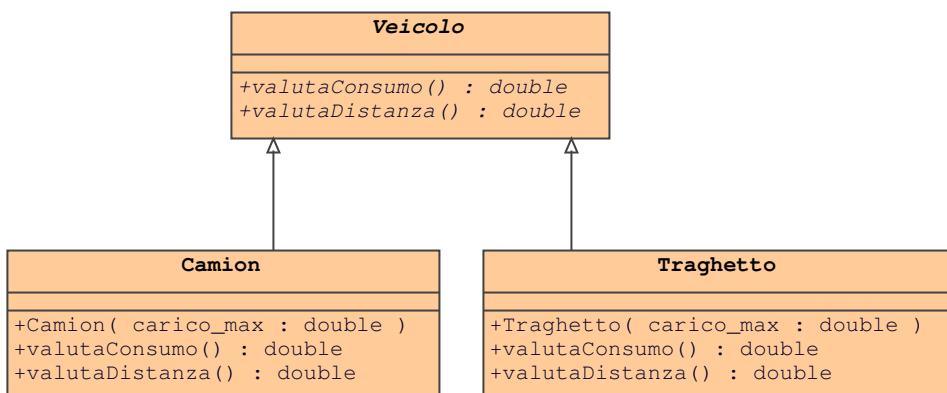
- La valutazione dei consumi tra i due tipi di veicoli potrebbe essere totalmente differente.
- Non ha senso che la classe Veicolo fornisca i due metodi valutaDistanza() e valutaConsumoPerKm(), ma le sue sottoclassi si.

LP1 – Lezione 13

7 / 44

Soluzione A.1

- Java permette ad una superclasse di dichiarare un metodo, del quale non fornirà una realizzazione.
- La realizzazione (*implementazione*) sarà fornita dalle sottoclassi.
- Tali metodi sono *astratti* e una classe con uno o più metodi astratti è una *classe astratta*.



- In UML ricordiamo che metodi o classi astratte si rappresentano utilizzando il carattere *corsivo*.
- 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` che `public` (perché?).

LP1 – Lezione 13

8 / 44

Soluzione A.2

```
public abstract class Veicolo {  
    public abstract double valutaConsumoPerKm();  
    public abstract double valutaDistanza();  
}
```

```
public class Camion extends Veicolo {  
    public Camion (double carico_max) {...}  
  
    public double valutaConsumoPerKm() {  
        // Restituisce il consumo/Km  
        // per un certo modello di camion  
    }  
    public double valutaDistanza() {  
        // Restituisce la distanza che deve  
        // percorrere per i viaggi  
    }  
}
```

LP1 – Lezione 13

9 / 44

Soluzione A.3

```
public class Traghetto extends Veicolo {  
    public Traghetto (double carico_max) {...}  
  
    public double valutaConsumoPerKm() {  
        // Restituisce il consumo/Km  
        // per un certo traghetto  
    }  
    public double valutaDistanza() {  
        // Restituisce la distanza che deve  
        // percorrere per i viaggi in mare  
    }  
}
```

LP1 – Lezione 13

10 / 44

Problema B

Riprendiamo la classe RapportoCarburante:

```
public class RapportoCarburante {  
    public void generaTesto(PrintStream output) {  
        ...  
        carburante = v.valutaConsumoPerKm() * v.valutaDistanza();  
        ...  
    }  
}
```

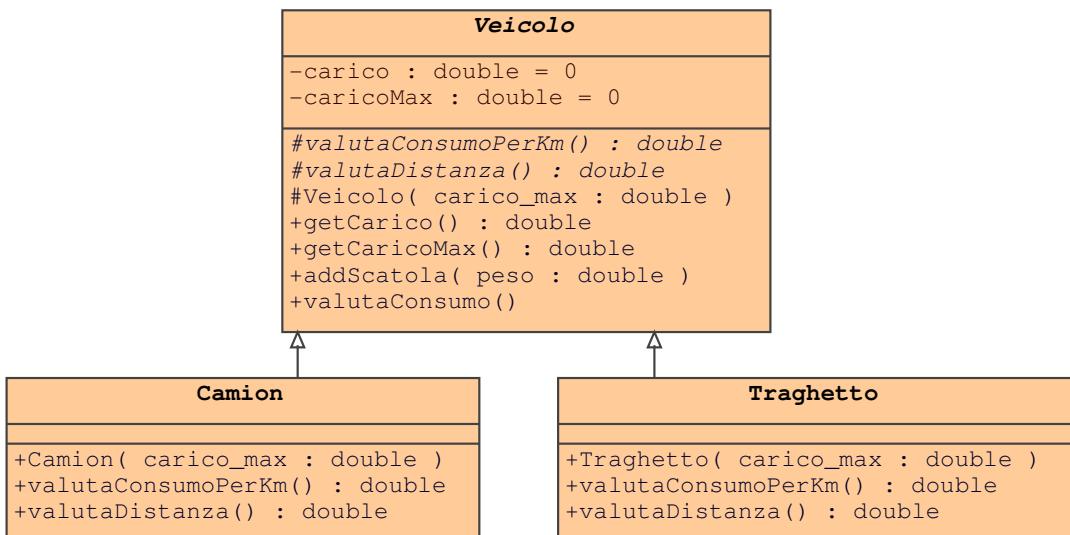
Il calcolo del consumo di un veicolo non dovrebbe essere eseguito in questa classe, ma dovrebbe essere una responsabilità del veicolo stesso, cioè appartenere alla classe Veicolo. Ma nella classe veicolo non ci sono le realizzazioni dei metodi `valutaConsumoPerKm()` e `valutaDistanza()`.

Nessun problema, perché l'invocazione virtuale dei metodi serve proprio a questo. Deleghiamo, pertanto, ad un veicolo il calcolo del consumo:

```
public class RapportoCarburante {  
    public void generaTesto(PrintStream output) {  
        ...  
        carburante = v.valutaConsumo();  
        ...  
    }  
}
```

Soluzione B

- La tecnica di soluzione consiste nel porre in una classe astratta un metodo concreto (nel nostro caso `valutaConsumo()`) che utilizzi metodi astratti nella stessa classe.
- Poiché tale tecnica è assai ricorrente, essa viene comunemente indicata come *Metodo Sagoma* (*Template Method Design Pattern*).



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.

LP1 – Lezione 13

14 / 44

Sintassi

- La sintassi è:

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

- Nota – una interfaccia può anche dichiarare costanti:

```
public static final int COSTANTE = 7;
```

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

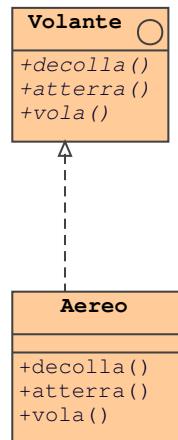
```
int COSTANTE = 7;
```

LP1 – Lezione 13

15 / 44

Esempio (1)

Immaginiamo un gruppo di oggetti che condividono la stessa abilità: essi volano. Si può costruire una interfaccia pubblica, chiamata Volante, che descriva tre operazioni: decolla, atterra e vola.

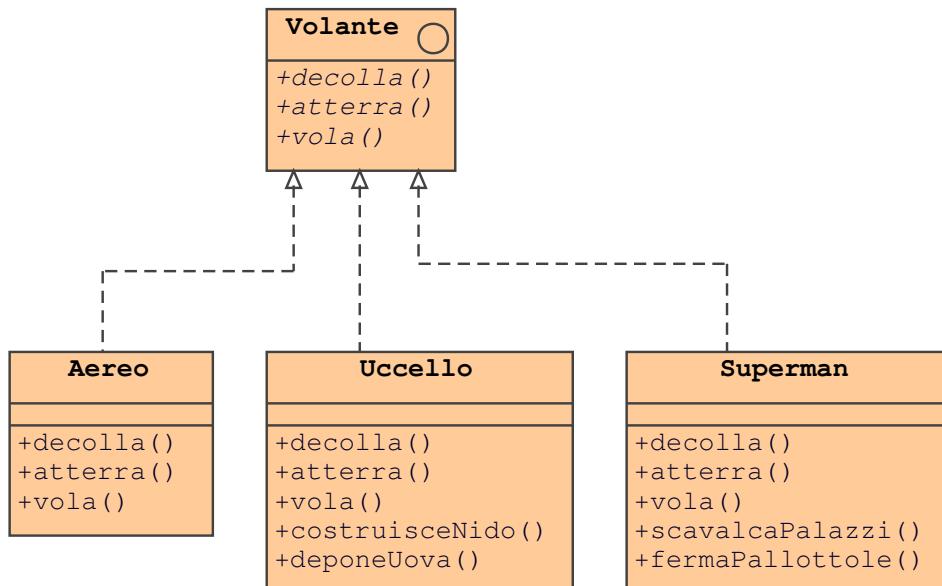


```
public interface Volante {
    public void decolla();
    public void atterra();
    public void vola();
}
```

```
public class Aereo
    implements Volante {
    public void decolla() {
        // accelera fino a sollevarsi
        // alza il carrello
    }
    public void atterra() {
        // abbassa il carrello
        // decelera e abbassa i flap
        // fino a toccare terra, poi
        // frena
    }
    public void vola() {
        // tieni il motore acceso
    }
}
```

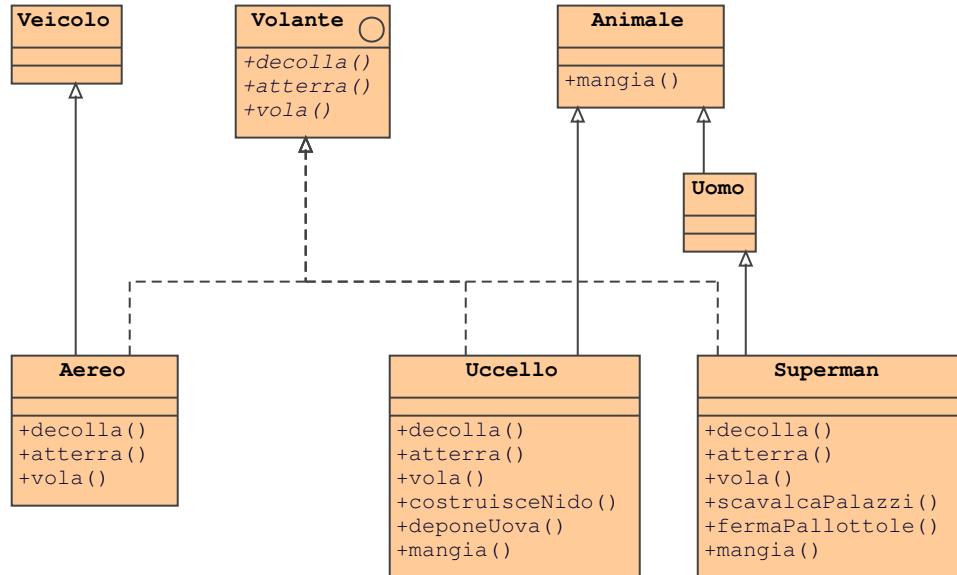
Esempio (2)

- Ma quali altri oggetti volano?
- Per esempio:



Esempio (3)

Qual è la superclasse di Uccello? Che cos'è un uccello?



Questo suona come ereditarietà multipla. Non proprio. Il pericolo dell'ereditarietà multipla è che una classe possa ereditare due distinte implementazioni dello stesso metodo. Ciò non è possibile con le interfacce.

LP1 – Lezione 13

18 / 44

Esempio (4)

Prendiamo per esempio la classe Uccello:

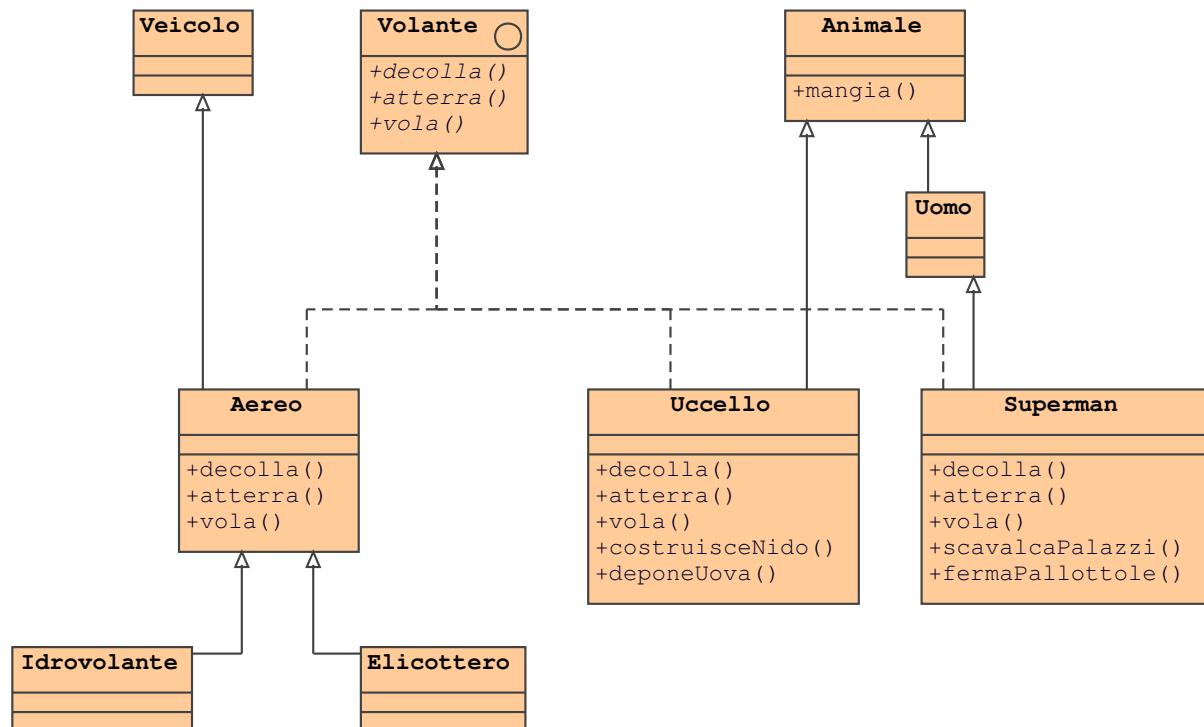
```
public class Uccello extends Animale implements Volante {
    public void decolla() {
        // implementazione di un metodo
    }
    public void atterra() {
        // implementazione di un metodo
    }
    public void vola() {
        // implementazione di un metodo
    }
    public void costruisceNido() {
        // competenza propria
    }
    public void deponeUova() {
        // competenza propria
    }
    public void mangia() {
        // sovrapposizione di un metodo
    }
}
```

LP1 – Lezione 13

19 / 44

Esempio (5)

Supponiamo di voler costruire un sistema software di controllo dei voli. Esso dovrà garantire i permessi di decollo e di atterraggio a qualunque tipo di oggetto volante.



LP1 – Lezione 13

20 / 44

Esempio (6)

Una classe che userà le classi precedenti potrebbe, per esempio, essere:

```
public class Aeroporto {
    public static void main(String[] args) {
        Aeroporto metropolis = new Aeroporto();
        Elicottero eli = new Elicottero();
        Idrovolante idro = new Idrovolante();
        Volante S = Superman.getSuperman();

        metropolis.daiPermessoAtterraggio(eli);
        metropolis.daiPermessoAtterraggio(idro);
        metropolis.daiPermessoAtterraggio(S);
    }

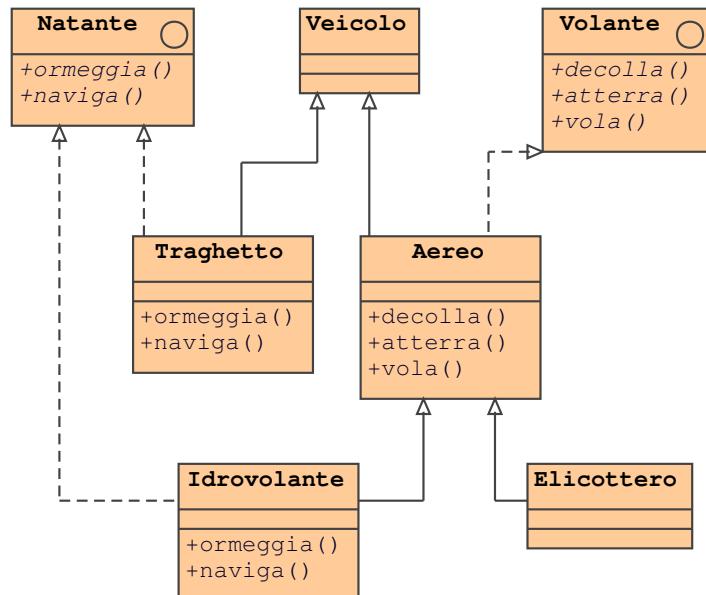
    private void daiPermessoAtterraggio(Volante v) {
        v.atterra();
    }
}
```

LP1 – Lezione 13

21 / 44

Esempio (7)

Una classe può implementare più di una interfaccia. Un idrovolante, non solo vola, ma anche naviga. La classe Idrovolante estende la classe Aereo, ed eredita l'implementazione dell'interfaccia Volante, ma implementa anche l'interfaccia Natante.



LP1 – Lezione 13

22 / 44

Esempio (8)

In quest'ultimo caso, una classe che userà le classi precedenti potrebbe essere:

```
public class Porto {
    public static void main(String[] args) {
        Porto portoNapoli = new Porto();
        Traghetto trag = new Traghetto();
        Idrovolante idro = new Idrovolante();

        portoNapoli.daiPermessoOrmeggio(trag);
        portoNapoli.daiPermessoOrmeggio(idro);
    }

    private void daiPermessoOrmeggio(Natante n) {
        n.ormeggia();
    }
}
```

LP1 – Lezione 13

23 / 44

Vantaggi delle interfacce

Le interfacce^a sono spesso considerate una alternativa all'ereditarietà multipla, anche se esse forniscono diversa 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, senza forzare una relazione tra classi;
- per simulare l'ereditarietà multipla, dichiarando una classe che implementi varie interfacce.

LP1 – Lezione 13

24 / 44

^aIl concetto di interfaccia è preso in prestito da Objective-C, in cui essa è chiamata *protocollo*.

Esercizi

25 / 44

Esercizi

1. Creare una gerarchia di animali con radice nella classe astratta *Animale*. Alcune delle classi dovranno implementare l'interfaccia *Domestico*.

LP1 – Lezione 13

25 / 44

Casting di riferimenti

26 / 44

Introduzione

- I riferimenti, come i primitivi, partecipano alla conversione automatica per assegnamento (e per passaggio di parametri) e al casting.
- Non c'è promozione aritmetica di riferimenti, poiché i riferimenti non possono essere operandi aritmetici.
- Nella conversione e nel casting di riferimenti 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, poiché il compilatore ha tutte le informazioni per determinare 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 tre tipi: quello dei due riferimenti e quello proprio dell'oggetto; strano, ma vero).

LP1 – Lezione 13

27 / 44

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:
 - ◆ ad una classe;

- ◆ ad una interfaccia;
- ◆ ad un array.

- La conversione può avvenire in:

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

	Oldtype è una classe	Oldtype è una interfaccia	Oldtype è un array
Newtype è una classe	Oldtype deve essere 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 due array devono essere riferimenti auto-convertibili

LP1 – Lezione 13

28 / 44

Conversioni automatiche (2)

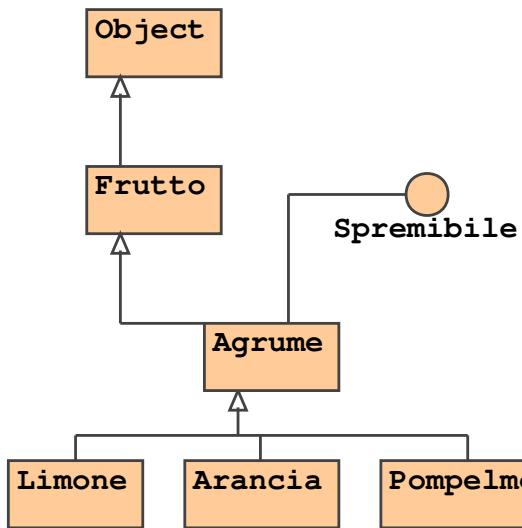
Regola breve:

- Un tipo interfaccia può essere convertito solo ad un tipo interfaccia o ad `Object`. Se il nuovo tipo è una interfaccia essa deve essere una superinterfaccia di quella del vecchio tipo.
- Un tipo classe può essere convertito ad un tipo classe o ad un tipo interfaccia. Se convertito ad un tipo classe, il nuovo tipo deve 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 un array di riferimenti (non di primitivi) può essere convertito ad un array, e il vecchio tipo degli elementi deve essere convertibile al nuovo tipo degli elementi.

LP1 – Lezione 13

29 / 44

Esempi (1)



```
Arancia arancia = new Arancia();
Agrume agrume = arancia; // OK
```

```
Agrume agrume = new Agrume();
Arancia arancia = agrume; // NO
```

```
Pompelmo p = new Pompelmo();
Spremibile s = p; // OK
Pompelmo p2 = s; // Errore comp.
```

Una interfaccia può solo essere auto-convertita ad Object.

```
Frutto frutta[];
Limone limoni[];
Agrume agrumi[] = new Agrume[10];
for (int i=0; i<10; i++) {
    agrumi[i] = new Agrume();
}
frutta = agrumi; // OK
limoni = agrumi; // Errore comp.
```

In questo caso contano i tipi dei riferimenti negli elementi degli array.

Esempi (2)

A proposito di overloading e overriding, in attinenza alle conversioni automatiche di riferimenti, ricordiamo che la scelta del metodo da associare ad una invocazione avviene

- in compilazione per i metodi sovraccaricati (basandosi sul tipo dei riferimenti nei parametri),
- in esecuzione per i metodi sovrapposti (basandosi sulla effettiva presenza e visibilità del metodo nell'oggetto).

Quindi:

Esempi (3)

```
class A {}  
class B extends A {  
    public void g(A a) {  
        System.out.println("A");  
    }  
}  
class UseAB extends B {  
    public void f(A a) {  
        System.out.println("A");  
    }  
    public void f(B a) {  
        System.out.println("B");  
    }  
    public void g(A b) {  
        System.out.println("B");  
    }  
  
    public static void main (  
        String[] args) {  
        UseAB u = new UseAB();  
        u.metodo();  
    }  
}
```

```
public void metodo() {  
    A a = new A();  
    B b = new B();  
    A x = b;           // auto-conv.  
  
    f(a);            // stampa A  
    f(b);            // stampa B  
    f(x);            // stampa A!!!  
    f(new B());      // stampa B  
  
    g(a);            // stampa B  
    g(b);            // stampa B  
    g(x);            // stampa B  
    g(new B());      // stampa B  
    super.g(a);      // stampa A  
    super.g(b);      // stampa A  
    super.g(x);      // stampa A  
}  
}
```

LP1 – Lezione 13

32 / 44

Casting (1)

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

- Un cast di ampliamento, la cui conversione sarebbe avvenuta anche senza di esso, viene autorizzato e non causa problemi né in compilazione, né 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 completamente e può accadere che, sebbene la conversione sia autorizzata come legale, essa fallisca in esecuzione.
- Infatti, nel casting esplicito, entrano in gioco tre tipi: quello dei due riferimenti (a sinistra e a destra dell'assegnazione) e quello vero (l'identità, l'imprinting di nascita) dell'oggetto.
- Esaminiamo quello che succede, passo per passo.

LP1 – Lezione 13

33 / 44

Casting (2)

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 una 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 impl. l'interfaccia o impl. Serializable	Errore di compilazione
Newtype è una interfaccia	Sempre OK	Oldtype deve impl. 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 due array devono essere <u>riferimenti</u> convertibili per cast

LP1 – Lezione 13

34 / 44

Casting (3)

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.

LP1 – Lezione 13

35 / 44

Casting (4)

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

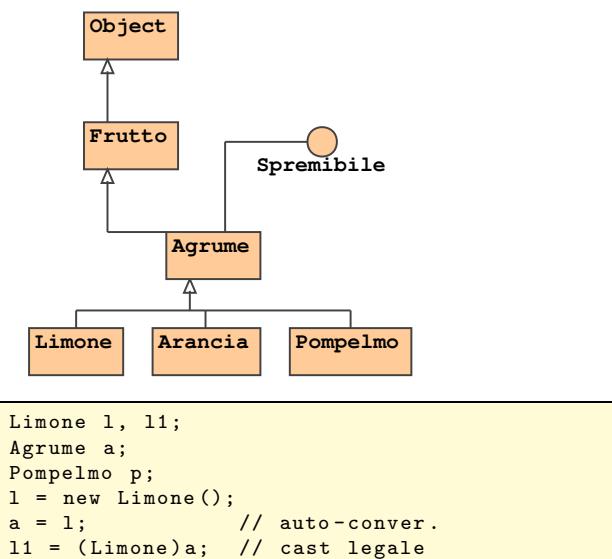
- Se Newtype è una classe, la classe originaria Objtype dell'oggetto deve essere Newtype oppure una sua sottoclasse.
- Se Newtype è una interfaccia, la classe originaria Objtype dell'oggetto deve implementare Newtype.

È tempo di fare alcuni esempi chiarificatori:

LP1 – Lezione 13

36 / 44

Esempi



```
p = (Pompelmo)a; // cast illeg.  
// err. esecuz.
```

```
Limone l, l1;  
Spremibile s;  
l = new Limone();  
s = l; // OK  
l1= s; // err. compil.  
l1= (Limone)s; // OK
```

Una interfaccia può essere auto-convertita solo ad Object.

```
Limone l[];  
Spremibile s[];  
Agrume a[];  
l = new Limone[7];  
s = l; // OK  
a = (Agrume[])s; // OK
```

È sempre lecito il cast di una interfaccia ad una classe non-final in compilazione. Qui il cast ha successo anche in esecuzione.

LP1 – Lezione 13

37 / 44

Questionario

38 / 44

D 1

Quale dei seguenti enunciati è corretto?

- A. Solo i tipi primitivi sono convertiti automaticamente; per cambiare il tipo di un riferimento bisogna fare un cast.
- B. Solo i riferimenti sono convertiti automaticamente; per cambiare il tipo di un primitivo bisogna fare un cast.
- C. La promozione aritmetica di riferimenti richiede il cast esplicito.
- D. Sia i tipi primitivi, sia i riferimenti possono essere convertiti sia automaticamente sia attraverso un cast.
- E. Il cast dei tipi numerici richiede un controllo in esecuzione.

LP1 – Lezione 13

39 / 44

D 2

Quale dei seguenti enunciati è vero?

- A. I riferimenti ad oggetti possono essere convertiti nelle assegnazioni e non nelle invocazioni di metodi.
- B. I riferimenti ad oggetti possono essere convertiti nelle invocazioni di metodi e non nelle assegnazioni.
- C. I riferimenti ad oggetti possono essere convertiti sia nelle assegnazioni, sia nelle invocazioni di metodi, ma le regole nei due casi sono diverse.
- D. I riferimenti ad oggetti possono essere convertiti sia nelle assegnazioni, sia nelle invocazioni di metodi, e le regole nei due casi sono identiche.
- E. I riferimenti ad oggetti non possono essere mai convertiti.

LP1 – Lezione 13

40 / 44

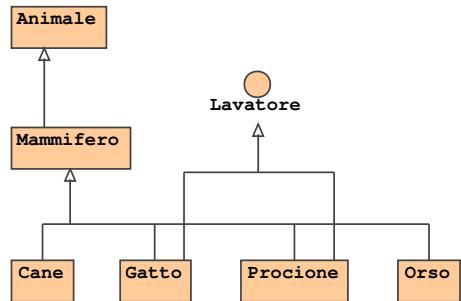
D 3

Quale linea del codice seguente non compila?

```
1. Object ob = new Object();
2. String stringarr[] = new String[50];
3. Float floater = new Float(3.14f);
4.
5. ob = stringarr;
6. ob = stringarr[5];
7. floater = ob;
8. ob = floater;
```

- A. La linea 5.
- B. La linea 6.
- C. La linea 7.
- D. La linea 8.

D 4



```
3.
4. billy = new Cane();
5. anim = billy;
6. fido = (Cane)anim;
```

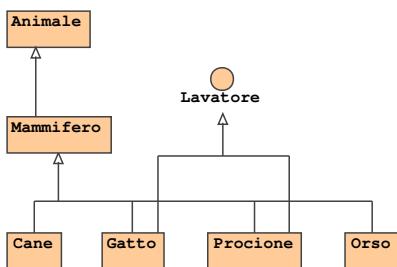
Quale dei seguenti enunciati è vero?

- A. La linea 5 non compila.
- B. La linea 6 non compila.
- C. Il codice è compilato correttamente ma viene lanciata una eccezione in esecuzione.
- D. Il codice è compilato ed eseguito correttamente.
- E. Il codice è compilato ed eseguito correttamente, ma il cast alla linea 6 è superfluo e può essere eliminato.

Si consideri il codice seguente:

```
1. Cane billy, fido;
2. Animale anim;
```

D 5



```
7. bubu = (Orso)lala;
```

Quale dei seguenti enunciati è vero?

- A. La linea 6 non compila poiché è richiesto un cast esplicito per convertire un Gatto ad un Lavatore.
- B. La linea 7 non compila poiché non si può fare il cast da una interfaccia ad una classe.
- C. Il codice è compilato ed eseguito correttamente ma il cast alla linea 7 non è necessario.
- D. Il codice è compilato ma, alla linea 7, lancia una eccezione in esecuzione poiché la conversione a runtime di una interfaccia in una classe è proibita.
- E. Il codice è compilato ma, alla linea 7, lancia una eccezione poiché il tipo di lala in esecuzione non può essere convertito al tipo Orso.

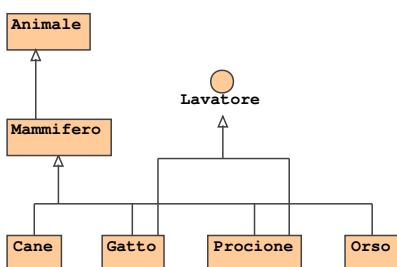
Si consideri il codice seguente:

```
1. Gatto fufi;  
2. Lavatore lala;  
3. Orso bubu;  
4.  
5. fufi = new Gatto();  
6. lala = fufi;
```

LP1 – Lezione 13

43 / 44

D 6



```
5. rocco = new Procione();  
6. lala = rocco;  
7. bubu = lala;
```

Quale dei seguenti enunciati è vero?

- A. La linea 6 non compila: è richiesto un cast esplicito.
- B. La linea 7 non compila: è richiesto un cast esplicito.
- C. Il codice è compilato ed eseguito correttamente.
- D. Il codice è compilato ma viene lanciata una eccezione alla linea 7, poiché in esecuzione la conversione da una interfaccia ad una classe è proibita.
- E. Il codice è compilato ma viene lanciata una eccezione alla linea 7, poiché la classe dell'oggetto lala non può essere convertita al tipo Orso.

Si consideri il codice seguente:

```
1. Procione rocco;  
2. Orso bubu;  
3. Lavatore lala;  
4.
```

LP1 – Lezione 13

44 / 44

Linguaggi di Programmazione I – Lezione 16

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

10 giugno 2008

Introduzione	3
Approccio ad una GUI	4
La mia prima GUI	5
Gestione degli eventi	6
La comunicazione tra sorgente e ascoltatore	7
GUI con un solo pulsante	8
Ascoltatori, sorgenti ed eventi	9
Proviamo con due pulsanti e due azioni	10
Un primo tentativo	11
Un altro tentativo	12
Soluzioni a disposizione	13
Sarebbe bello se...	14
Utilità delle classi interne	15
Classe membro di altra classe	16
Introduzione	17
Sintassi base	18
Costruzione (1)	19
Costruzione (2)	20
Costruzione (3)	21
Costruzione (4)	22
Esempio 1	23
Esempio 2: GUI a due pulsanti	24
Modificatori (1)	25
Modificatori (2)	26
Classe definita all'interno di un metodo	27
Introduzione	28
Accesso a var. locali	29
Esempio 3	30
Classi anonime	31
Introduzione	32
Esempio 4.a	33
Esempio 4.b	34

Questionario	35
D 1	36
D 2	37
D 3	38
D 4	39
D 5	40
D 6	41
D 7	42
D 8	43
D 9	44
D 10	45

Classi interne

Introduzione

Classe membro di altra classe

Classe definita all'interno di un metodo

Classi anonime

Questionario

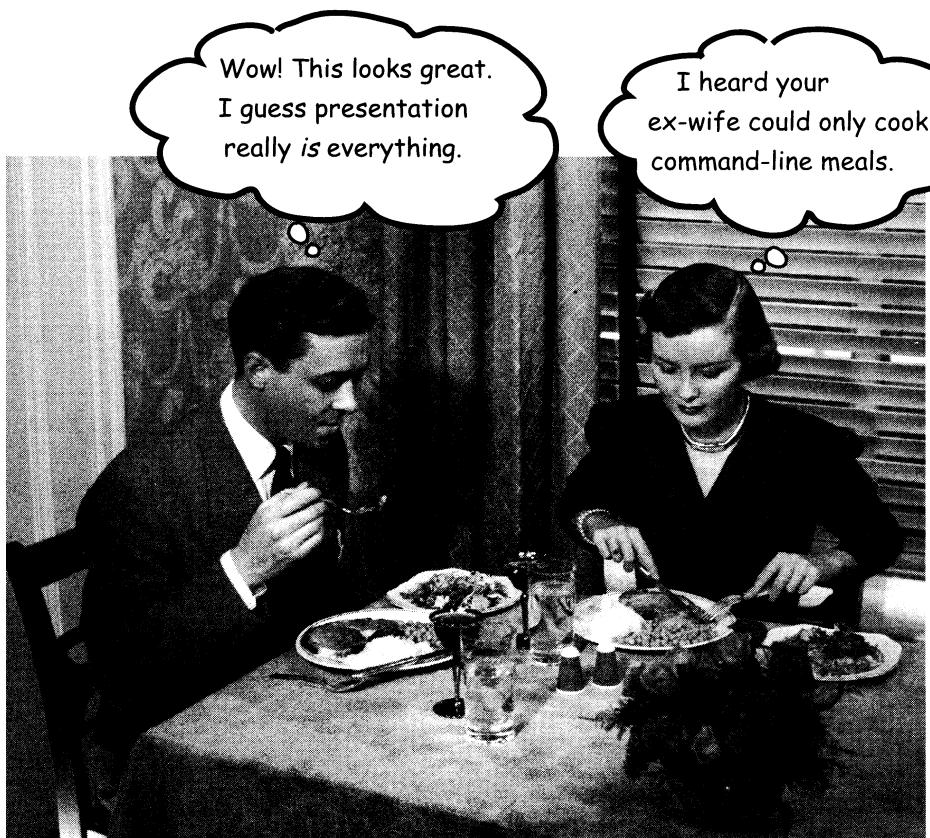
LP1 – Lezione 16

2 / 45

Introduzione

3 / 45

Approccio ad una GUI



LP1 – Lezione 16

4 / 45

La mia prima GUI

```
import javax.swing.*;  
  
public class ProvaGUI {  
    private JFrame frame;  
    private JButton pulsante;  
  
    public static void main (String[] args) {  
        ProvaGUI gui = new ProvaGUI();  
        gui.go();  
    }  
  
    public void go() {  
        frame = new JFrame();  
        pulsante = new JButton("Cliccami");  
  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        frame.getContentPane().add(pulsante);  
        frame.setSize(300,300);  
        frame.setVisible(true);  
    }  
}
```

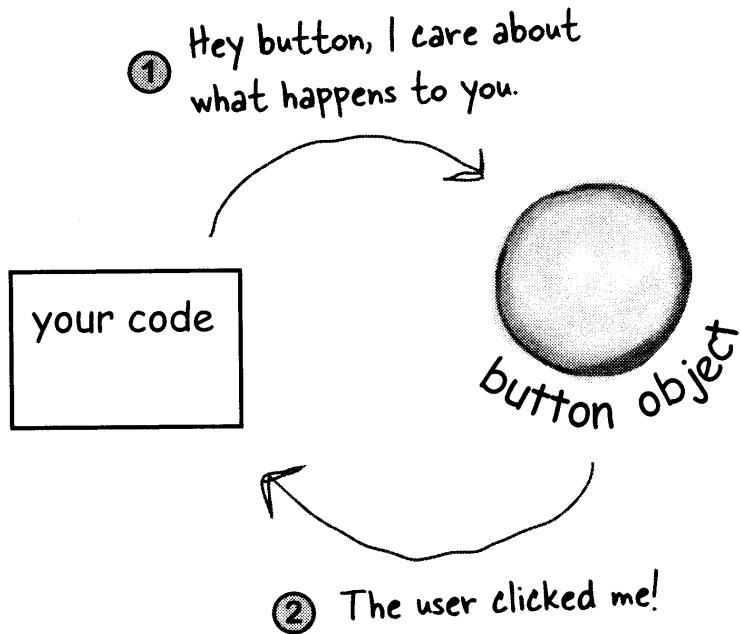
Ma non c'è nessuna azione collegata alla pressione del pulsante.

LP1 – Lezione 16

5 / 45

Gestione degli eventi

- Piuttosto che controllare ciclicamente e continuamente lo stato dell'oggetto pulsante, è molto più efficiente lasciare che sia l'oggetto pulsante a comunicarci il proprio cambiamento di stato:

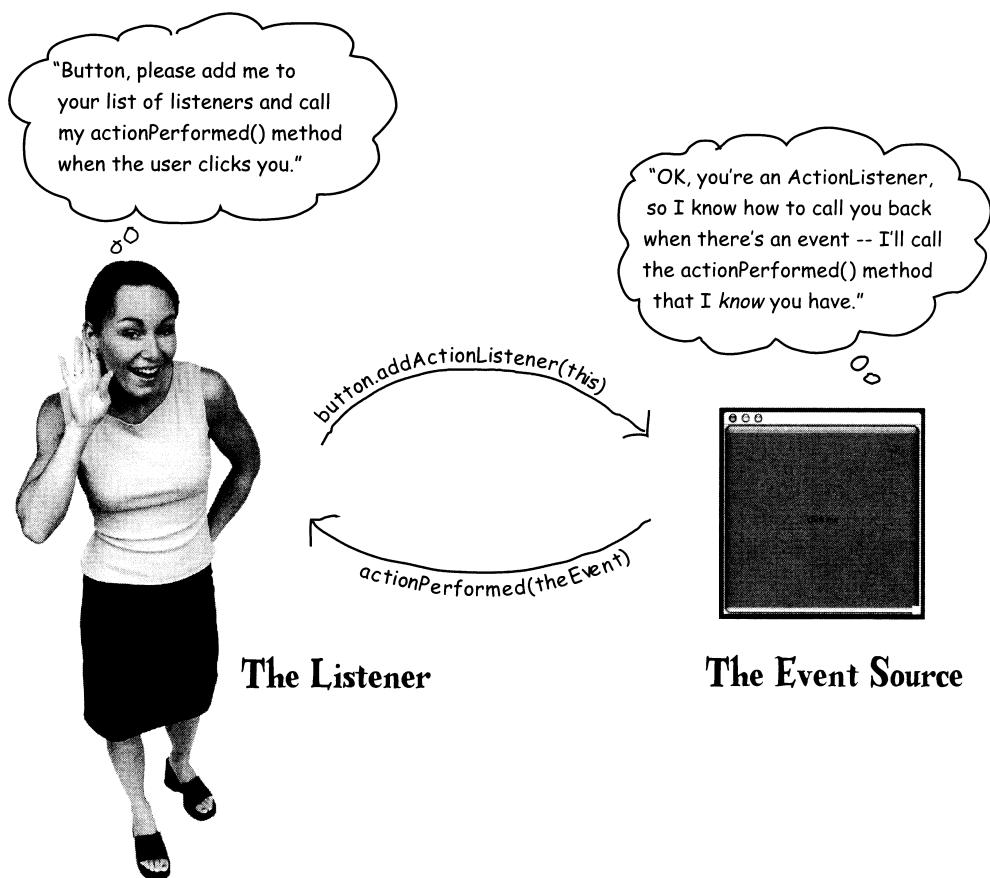


- Il nostro codice diventa un *ascoltatore* e il pulsante è la *sorgente*.

LP1 – Lezione 16

6 / 45

La comunicazione tra sorgente e ascoltatore



GUI con un solo pulsante

```
import javax.swing.*;
import java.awt.event.*;

public class Gui1 implements ActionListener {
    private JFrame frame; private JButton pulsante;

    public static void main (String[] args) {
        new Gui1().go();
    }

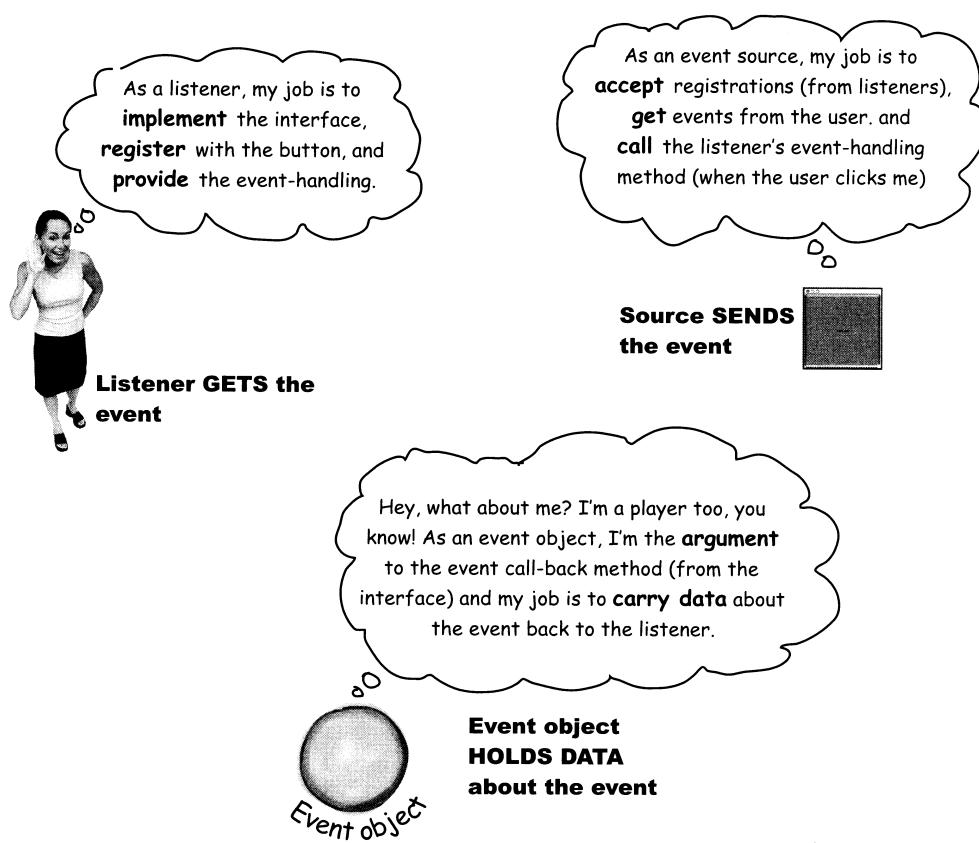
    public void go() {
        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        pulsante = new JButton("Cliccami");
        pulsante.addActionListener(this);

        frame.getContentPane().add(pulsante);
        frame.setSize(300,300); frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        pulsante.setText("Pulsante cliccato!");
    }
}
```

Ascoltatori, sorgenti ed eventi



Proviamo con due pulsanti e due azioni

```
public class Gui2 implements ActionListener {  
    private JFrame frame; private JButton pulsante1, pulsante2;  
    // qui il solito codice come prima:  
    // - generazione dell'oggetto frame e dei due oggetti  
    // pulsanti;  
    // - registrazione di this con i due oggetti pulsanti  
    //  
    // ora devo descrivere le due diverse azioni:  
  
    public void actionPerformed(ActionEvent event) {  
        pulsante1.setText("P1: cliccato!");  
    }  
  
    public void actionPerformed(ActionEvent event) {  
        pulsante2.setText("P2: cliccato!");  
    }  
}
```

Ma questo è un sovraccaricamento illegale di metodi!

LP1 – Lezione 16

10 / 45

Un primo tentativo

- A questo punto, potrei definire un unico metodo che chieda all'evento ricevuto come parametro informazioni sulla natura della sorgente e che si comporti in modo diverso in base alla risposta,
- ma ciò significherebbe costruire codice non ben separato: una modifica del funzionamento della sorgente comporterebbe la modifica di ogni ascoltatore.

LP1 – Lezione 16

11 / 45

Un altro tentativo

Creazione di due classi ActionListener separate:

```
public class Gui2 {  
    private JFrame frame; private JButton pulsante1, pulsante2;  
    void go() {  
        // codice per istanziare i due oggetti ascoltatori  
        // e registrarli con i due oggetti pulsanti  
    }  
}
```

```
class Ascoltatore1 implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        pulsante1.setText("P1: cliccato!");  
    }  
}
```

```
class Ascoltatore2 implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        pulsante2.setText("P2: cliccato!");  
    }  
}
```

Ma pulsante1 e pulsante2 non sono accessibili all'esterno di Gui2.

LP1 – Lezione 16

12 / 45

Soluzioni a disposizione

- Rompere l'incapsulazione e rendere pubblici i riferimenti ai pulsanti nella classe Gui2,
- oppure rendere più complicate le classi degli ascoltatori, aggiungendo un attributo che possa riferire al pulsante ed un costruttore che consenta alla classe Gui2 di inizializzare quell'attributo con il valore del proprio attributo (privato) pulsante1 o pulsante2,
- oppure ...

LP1 – Lezione 16

13 / 45

Sarebbe bello se...



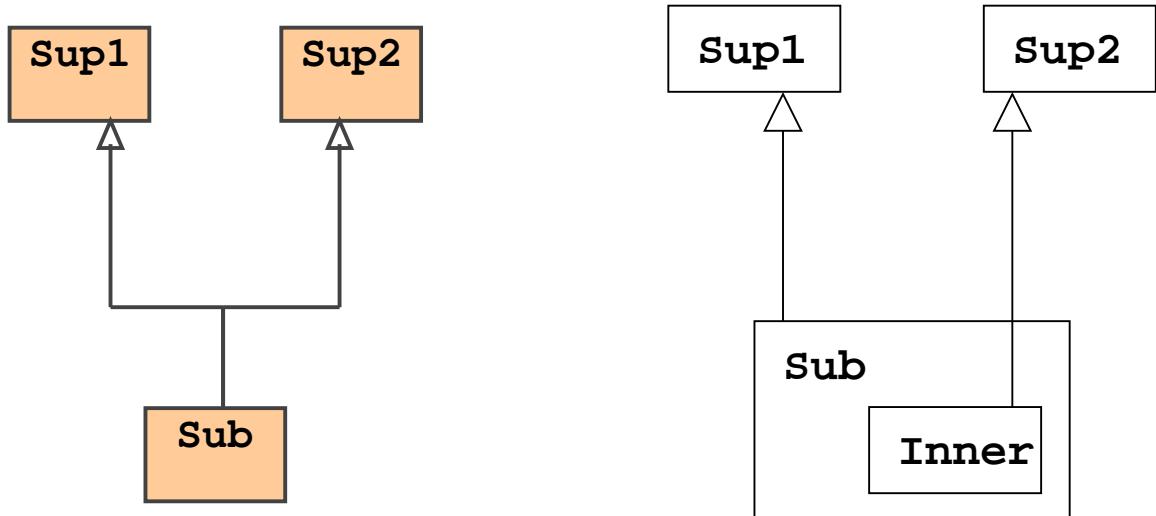
Wouldn't it be wonderful if you could have two different listener classes, but the listener classes could access the instance variables of the main GUI class, almost as if the listener classes belonged to the other class. Then you'd have the best of both worlds. Yeah, that would be dreamy.
But it's just a fantasy...

LP1 – Lezione 16

14 / 45

Utilità delle classi interne

- Classi *inner* sono state aggiunte con la JDK 1.1.
- Sono a volte chiamate classi annidate.
- Danno al programma maggiore chiarezza e lo rendono più conciso.
- Offrono anche una soluzione elegante al problema realizzativo dell'ereditarietà multipla:



LP1 – Lezione 16

15 / 45

Classe membro di altra classe

16 / 45

Introduzione

- Fondamentalmente, sono come le altre classi, ma sono dichiarate all'interno di altre classi.
- Possono essere poste in qualunque blocco, incluso i blocchi dei metodi.
- Classi definite all'interno di metodi differiscono leggermente dalle altre e saranno trattate meglio in seguito.
- Per ora, con il nome di “classe membro”, intenderemo una classe che non è definita in un metodo, ma semplicemente in un'altra classe (come gli altri membri: attributi e metodi).
- La complessità dell'argomento è relativa agli ambiti di validità e di accesso, in particolare all'accesso a variabili nell'ambito includente.

LP1 – Lezione 16

17 / 45

Sintassi base

```
public class Esterno {  
    private int x;  
    public class Interno {  
        private int y;  
        public void metodoInterno() {  
            System.out.println("y=" + y);  
        }  
    }  
    public void metodoEsterno() {  
        System.out.println("x=" + x);  
    }  
    // altri metodi ...  
}
```

- Il nome della classe includente diventa parte del nome della classe inclusa.

- In questo caso, i nomi completi delle due classi sono `Esterno` ed `Esterno.Interno`.
- Questo formato è reminiscente del modo in cui una classe è nominata all'interno di un pacchetto. Infatti, una classe interna appartiene alla propria classe includente allo stesso modo con cui una classe appartiene al proprio pacchetto.
- È illegale che una classe ed un pacchetto abbiano lo stesso nome, perciò non c'è ambiguità di interpretazione.
- Attenzione: la rappresentazione puntata vale solo all'interno del sorgente Java; essa non riflette il nome del file delle singole classi. Sul disco la classe interna si chiama `Esterno$Interno`.

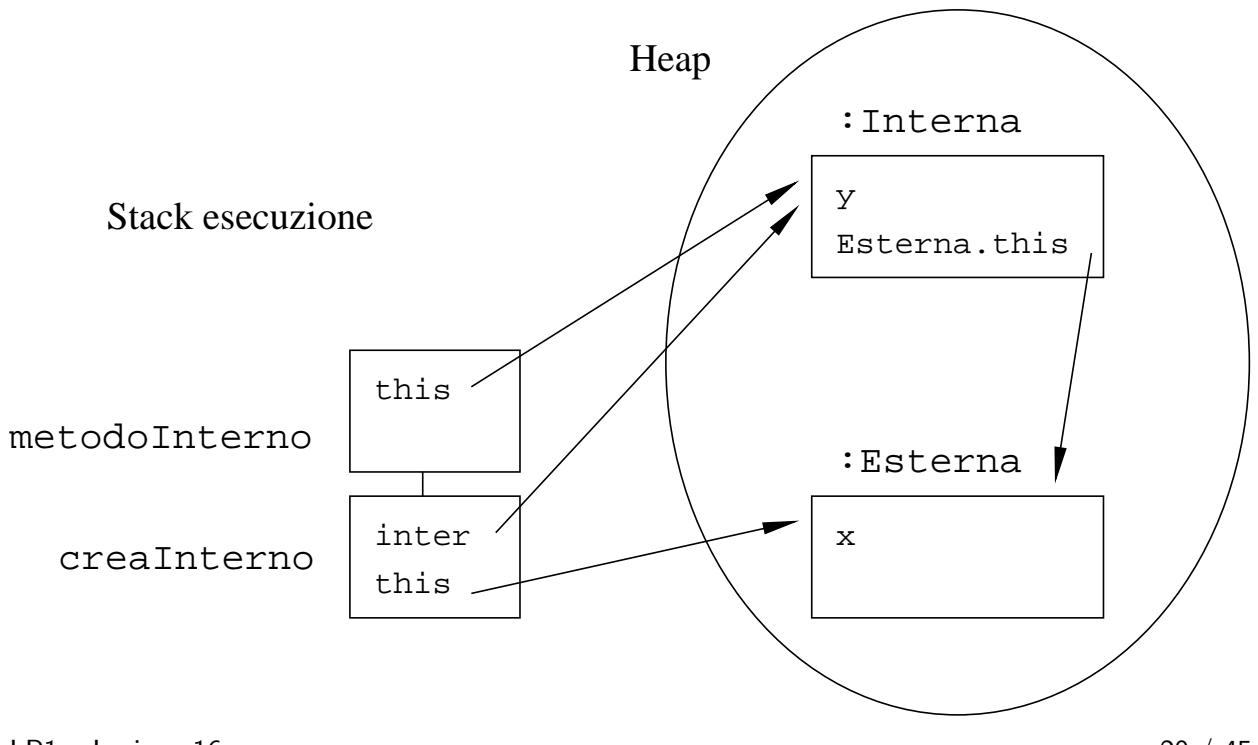
Costruzione (1)

```
public class Esterna {  
    private int x;  
    public class Interna {  
        private int y;  
        public void metodoInterno() {  
            System.out.println("x=" + x);  
            // attributo esterno  
            // accessibile anche  
            // con Esterna.this.x  
            System.out.println("y=" + y);  
            // attributo interno  
        }  
    }  
    public void metodoEsterno() {  
        System.out.println("x=" + x);  
    }  
    public void creaInterno() {  
        Interna inter = new Interna();  
        inter.metodoInterno();  
    }  
}
```

```
// altri metodi ...  
}
```

- Quando si deve costruire 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'oggetto esterno, avendo costruito quello interno, ne possiede un riferimento. L'oggetto interno, però, può accedere all'oggetto esterno attraverso il riferimento implicito (tale riferimento può essere anche esplicitato; nel nostro esempio si chiama `Esterna.this`): *oggetti interno ed esterno si appartengono*.

Costruzione (2)



LP1 – Lezione 16

20 / 45

Costruzione (3)

- Nel caso in cui non esista l'oggetto esterno, per esempio perché la JVM sta eseguendo un metodo statico, allora bisogna costruirne uno “al volo”:

```
public static void main(String args[]) {  
    Esterna.Interna i = new Esterna().new Interna();  
    i.metodoInterno();  
}
```

oppure, in forma meno compatta:

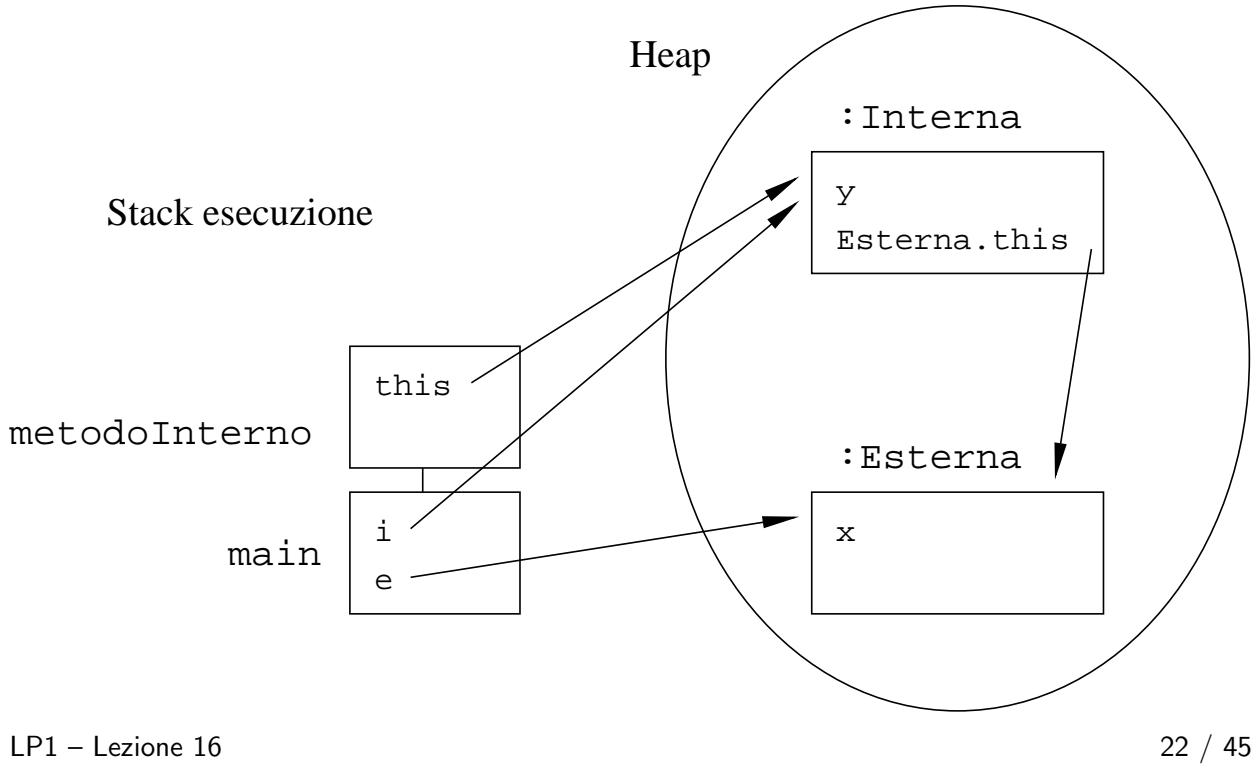
```
public static void main(String args[]) {  
    Esterna e = new Esterna();  
    Esterna.Interna i = e.new Interna();  
    i.metodoInterno();  
}
```

- In entrambi i casi `new` è usato come se fosse un metodo della classe esterna.

LP1 – Lezione 16

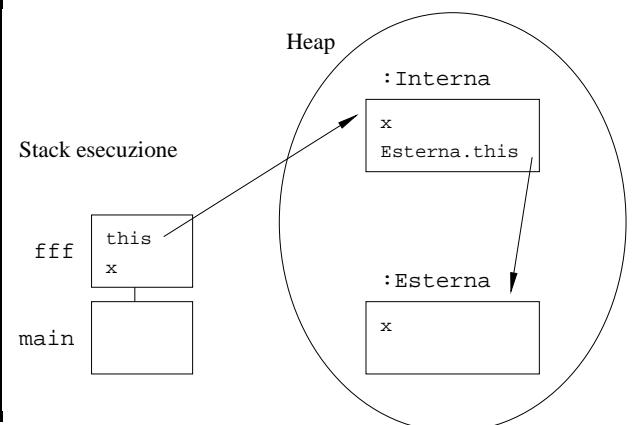
21 / 45

Costruzione (4)



Esempio 1

```
public class Esterna {  
    private int x;  
  
    public class Interna {  
        private int x;  
  
        public void fff(int x) {  
            // parametro locale:  
            x++;  
            // attributo interno:  
            this.x++;  
            // attributo esterno:  
            Esterna.this.x++;  
        }  
    }  
}
```



Esempio 2: GUI a due pulsanti

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Gui2 {
    private JFrame frame;
    private JButton pulsante1, pulsante2;

    public static void main (String[] args) {
        new Gui2().go();
    }

    public void go() {
        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        pulsante1 = new JButton("Pulsante 1");
        pulsante1.addActionListener(new Ascoltatore1());

        pulsante2 = new JButton("Pulsante 2");
        pulsante2.addActionListener(new Ascoltatore2());
    }
}

class Ascoltatore1 implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        pulsante1.setText("P1: cliccato!");
    }
}

class Ascoltatore2 implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        pulsante2.setText("P2: cliccato!");
    }
}
```

LP1 – Lezione 16

24 / 45

Modificatori (1)

- Classi membro possono essere optionalmente marcate con qualunque modificatore di accesso (a differenza delle classi top-level, che possono essere optionalmente marcate solo `public`). Il significato è identico a quello degli altri membri della classe (attributi o metodi). Per 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;
- ◆ sono praticamente delle classi top-level, con uno schema di nomi modificato;
- ◆ sono istanziate in questo modo:

```
public class Esterna {
    public static class Interna {}
    public static void main
        (String[] args) {
        Interna i =
            new Esterna.Interna();
    }
}
```

LP1 – Lezione 16

25 / 45

Modificatori (2)

Vogliamo esplicitare che esiste una importante restrizione ai modificatori attribuibili ai membri di una classe interna (attenzione, non ai membri di una classe top-level):

- Poiché una istanza di una classe interna non-static può solo esistere insieme ad una istanza della classe includente, cioè, in altri termini, poiché una classe interna non-static esiste solo per generare istanze e non per fornire servizi, allora
I MEMBRI DI UNA CLASSE INTERNA NON-static NON POSSONO ESSERE MARCATI static.
- Ciò implica, per esempio, che:
 - ◆ possono esistere classi static di livello di annidamento diverso dal primo solo se sono annidate in altre classi static;
 - ◆ è possibile che esistano interfacce annidate, ma in questo caso esse hanno l'implicito (o esplicito) modificatore static e seguono la stessa regola precedente.

LP1 – Lezione 16

26 / 45

Classe definita all'interno di un metodo

27 / 45

Introduzione

Classi definite in un metodo:

1. Come tutte le altre variabili del metodo, sono locali in quel metodo e non possono essere marcate con nessun modificatore tranne che con abstract o final (non entrambi, perché?).
2. Esse non sono accessibili (come tutte le altre variabili locali) in alcun modo all'esterno del metodo.
3. Hanno accesso limitato alle altre variabili locali del metodo (discuteremo subito).

LP1 – Lezione 16

28 / 45

Accesso a variabili locali

- La regola di accesso è semplice: *ogni variabile locale (anche un parametro) 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), poiché gli oggetti sullo heap possono sopravvivere ai record sullo stack.
- Sveliamo il trucco utilizzato dalla JVM per permettere l'accesso limitatamente alle variabili final:

La variabile final è una costante nel metodo. L'oggetto che deve accedervi ne possiede semplicemente una copia (per esempio generata durante la costruzione dell'oggetto). In tal modo, alla terminazione del metodo, la costante sarà ancora utilizzabile, anche se non esiste più il record di attivazione che la contiene.

LP1 – Lezione 16

29 / 45

Esempio 3

```
public class Esterna {
    private int m = (int)(Math.random() * 100);
    public static void main(String[] args) {
        Esterna oggetto = new Esterna();
        oggetto.fai(m, 2*m);
    }

    public void fai(int x, final int y) {
        int a = x + y;
        final int b = x - y;
        class Interna {
            public void metodo() {
                System.out.println("m=" + m);
//                System.out.println("x=" + x); // Illegale
                System.out.println("y=" + y);
//                System.out.println("a=" + a); // Illegale
                System.out.println("b=" + b);
            }
        }
        Interna oggetto = new Interna();
        oggetto.metodo();
    }
}
```

Classi anonime

Introduzione

- Possono essere usate per estendere un'altra classe oppure, in alternativa, per implementare una singola interfaccia.
- La sintassi non concede di fare entrambe le cose, né di implementare più di una interfaccia: se si dichiara una classe che implementa una singola interfaccia, allora la classe è sottoclassificata di `java.lang.Object`.
- Poiché 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.

Esempio 4.a

```
1. class A {  
2.     public void f() {  
3.         System.out.println("A");  
4.     }  
5. }  
6. class B {  
7.     A a = new A() {  
8.         public void f() {  
9.             System.out.println("B");  
10.        }  
11.    }; // notare il ';'  
12. }
```

- La variabile a si riferisce non ad una istanza di A, ma

ad una istanza di una sottoclasse anonima di A.

- La linea 7 comincia con una dichiarazione di variabile-riferimento ad A. Ma, invece di essere:

```
A a = new A(); // con ';
```

c'è una parentesi graffa aperta alla fine, che si chiude alla linea 11. Qui c'è il punto-e-virgola che manca.

- Quello che c'è tra la linea 7 e la 11 si può parafrasare: "Dichiara una variabile a di tipo A; poi dichiara una nuova classe, senza nome, *sottoclasse* di A, che contenga solo una sovrapposizione del metodo f(); infine, costruisci una istanza di tale sottoclasse ed assegna il suo riferimento ad a".

LP1 – Lezione 16

33 / 45

Esempio 4.b

Il polimorfismo è in azione: noi usiamo un riferimento ad una superclasse per riferirci ad una istanza di una sottoclasse. Ciò implica:

1. non avremo mai la possibilità di accedere a ciò che si aggiunge nella definizione della sottoclasse anonima

(con quale nome faremmo un cast?);

2. l'uso di una classe interna anonima è possibile solo per quanto riguarda i metodi che essa sovrappone, e che sono citabili solo perché presenti nella superclasse;
3. essa non può avere un costruttore esplicito (che nome avrebbe?).

```
class A {  
    public void f() {...}  
}  
class B {  
    A a = new A() {  
        public void g() {...}  
        public void f() {...}  
    };  
    public void usa() {  
        a.f(); // OK, f() e' anche metodo di A  
        a.g(); // Illegale, g() non e' un metodo di A  
    }  
}
```

LP1 – Lezione 16

34 / 45

Questionario

35 / 45

D 1

Quali, tra i seguenti enunciati, sono veri?

- A. Una classe interna può essere marcata private.
- B. Una classe interna può essere marcata static.
- C. Una classe interna definita in un metodo deve sempre essere anonima.
- D. Una classe interna definita in un metodo può accedere tutte le variabili locali di quel metodo.
- E. La costruzione di una classe interna può richiedere una istanza della classe esterna.

LP1 – Lezione 16

36 / 45

D 2

Si consideri la seguente definizione:

```
1. public class Outer {  
2.     public int a = 1;  
3.     private int b = 2;  
4.     public void method(final int c) {  
5.         int d = 3;  
6.         class Inner {  
7.             private void iMethod(int e) {  
8.                 }  
9.             }  
10.        }  
11.    }  
12. }
```

Quali variabili possono essere correttamente utilizzate alla linea 8?

- A. a
- B. b
- C. c
- D. d
- E. e

D 3

Quali dei seguenti enunciati sono veri?

- A. Sia Inner una classe non-static dichiarata all'interno di una classe pubblica Outer. Una istanza di Inner può essere costruita in questo modo:

```
new Outer().new Inner()
```

- B. Se una classe anonima all'interno della classe Outer è definita in modo da implementare l'interfaccia Rotante, essa può essere costruita così:

```
new Outer().new Rotante()
```

- C. Sia Inner una classe non-static dichiarata all'interno di una classe pubblica Outer. In un

metodo statico, una istanza di Inner può essere costruita in questo modo:

```
new Inner()
```

- D. Una istanza di classe anonima che implementa l'interfaccia Inter può essere costruita e restituita da un metodo come questo:

```
return new Inter(int x) {  
    int x;  
    public Inter(int x) {  
        this.x = x;  
    }  
};
```

D 4

Qual è il risultato del seguente codice?

```
public class MiaClasse {
    public static void main
        (String [] args) {
            Esterna e = new Esterna();
            System.out.println(
                e.creaInterna().getSegreto());
        }
}
class Esterna {
    private int secreto;
    Esterna() { secreto=7; }

    class Interna {
        int getSegreto() {
            return secreto; }
    }
}
```

```
Interna creaInterna () {
    return new Interna(); }
```

- A. Il codice non viene compilato, poiché la classe Interna non può essere dichiarata nella classe Esterna.
- B. Il codice non viene compilato, poiché il metodo creaInterna() non può passare oggetti della classe interna a metodi all'esterno della classe Esterna.
- C. Il codice non viene compilato, poiché la variabile secreto non è accessibile.
- D. Il codice non viene compilato, poiché il metodo getSegreto() non è visibile dal main.
- E. Il codice viene compilato correttamente e stampa 7.

LP1 – Lezione 16

39 / 45

D 5

Quali delle seguenti affermazioni riguardanti le classi interne sono vere?

- A. Una istanza di una classe interna static è associata ad una istanza di una classe esterna.
- B. Una istanza di una classe interna static può contenere membri non-static.
- C. Una interfaccia interna static può contenere membri non-static.
- D. Una interfaccia interna static è associata ad una istanza di classe esterna.
- E. Per ciascuna istanza di classe esterna, possono esistere molte istanze di una classe interna non-static.

LP1 – Lezione 16

40 / 45

D 6

```
public class Nesting {
    public static void main
        (String [] args) {
            B.C o = new B().new C();
        }
}
class A {
    int val;
    A(int v) { val = v; }
}
class B extends A {
    int val = 1;
    B() { super(2); }
    class C extends A {
        int val = 3;
        C() {
            super(4);
            System.out.print(B.this.val);
            System.out.print(C.this.val);
            System.out.print(super.val);
        }
    }
}
```

Qual è l'output del programma precedente?

- A. Errore di compilazione
- B. 234
- C. 142
- D. 134
- E. 321

LP1 – Lezione 16

41 / 45

D 7

Quali delle seguenti affermazioni sono vere?

- A. Classi interne non-static devono avere accessibilità public o di default.
- B. Tutte le classi interne possono contenere classi static.
- C. I metodi in tutte le classi interne possono essere dichiarati static.
- D. Tutte le classi interne possono essere dichiarate static.
- E. Le classi interne static possono contenere metodi non-static.

LP1 – Lezione 16

42 / 45

D 8

Quali delle seguenti affermazioni sono vere?

- A. Non si possono dichiarare membri static all'interno di classi interne non-static.
- B. Se una classe interna non-static è posta nella classe Esterna, allora i metodi di quella classe devono usare il prefisso Esterna.this per accedere ai membri della classe Esterna.
- C. Tutti le variabili di una classe interna devono essere dichiarate final.
- D. Classi anonime non possono avere costruttori.
- E. Se objRef è una istanza di una classe interna nella classe Esterna, allora (objRef instanceof Esterna) ha valore true.

LP1 – Lezione 16

43 / 45

D 9

Quali affermazioni sono vere?

- A. Classi membro di pacchetto possono essere dichiarate static.
- B. Classi membro di classi top-level possono essere dichiarate static.
- C. Classi locali possono essere dichiarate static.
- D. Classi anonime possono essere dichiarate static.
- E. Nessuna classe può essere dichiarata static.

LP1 – Lezione 16

44 / 45

D 10

```
interface I { int f(); }
```

Data la dichiarazione precedente, quali dei seguenti metodi sono validi?

A.

```
I makeI(int i) {
    return new I() {
        public f() {
            return i;
        }
    };
}
```

B.

```
I makeI(final int i) {
    return new I {
        public f() {
            return i;
        }
    };
}
```

C.

```
I makeI(int i) {
    class MyI implements I {
        public int f() {
            return i;
        }
    }
    return new MyI();
}
```

D.

```
I makeI(final int i) {
    class MyI implements I {
        public int f() {
            return i;
        }
    }
    return new MyI();
}
```

Linguaggi di Programmazione I – Lezione 14

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

29 maggio 2008

String e StringBuffer	3
String (1)	4
String (2)	5
Esempi	6
String (3)	7
StringBuffer (1)	8
Note importanti	9
Garbage collection	10
Ancora?.....	11
Ancora??	12
Ancora???	13
Mo' basta!	14
Questionario	15
D 1.....	16
D 2.....	17
D 3.....	18

Complementi

String e StringBuffer

Garbage collection

Questionario

LP1 – Lezione 14

2 / 18

String e StringBuffer

3 / 18

String (1)

- Le stringhe sono oggetti che possono essere creati come segue:

```
String s1 = new String();  
s = "abcdef";
```

oppure così:

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

oppure ancora così:

```
String s = "abcdef";
```

- Vedremo tra poco quali sono le (sottili) differenze tra questi modi.
- La cosa importante da capire è che le stringhe sono oggetti **immutabili**.

LP1 – Lezione 14

4 / 18

String (2)

- Immutabilità significa che, una volta assegnato all'oggetto un valore, esso è fissato per sempre (capiremo tra poco anche perché deve essere così).
- Attenzione: immutabili sono gli oggetti **String**, non i loro riferimenti. Questi ultimi possono cambiare valore.

LP1 – Lezione 14

5 / 18

Esempi

```
String s = "Marcello";
String s2 = s;
s = s.concat(" Sette");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Marcello";
x.concat(" Sette");
System.out.println(x);    // cosa stampa?
```

```
String s1 = "A";
String s2 = s1 + "B";
s1.concat("C");
s2.concat(s1);
s1 += "D";                // Quanti oggetti in gioco?
System.out.println(s1 + s2); // Cosa stampa?
```

```
String s1 = "abc";
String s2 = s1 + "";
String s3 = "abc";
System.out.println(s1 == s2); // false!
System.out.println(s1 == s3); // true!
```

String (3)

- 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 è 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.
- Ora si capisce perché gli oggetti String devono essere immutabili. 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.
- Qual è quindi la (sottile) differenza tra i due enunciati?

```
String s = "abcdef";
```

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

StringBuffer (1)

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

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

- Attenzione:

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

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

LP1 – Lezione 14

8 / 18

Note importanti

- ATTENZIONE: mentre la classe `String` sovrappone il metodo `equals` in modo da controllare l'uguaglianza del contenuto dei due 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.

LP1 – Lezione 14

9 / 18

Garbage collection

10 / 18

Ancora?

```
public class TestGC {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Ciao");
        System.out.println(sb);
        // l'oggetto riferito da sb non e' ancora
        // eleggibile per GC
        sb = null;
        // ora e' eleggibile .
    }
}
```

Ma questo non è l'unico modo di rendere eleggibile un oggetto!

LP1 – Lezione 14

11 / 18

Ancora??

```
public class TestGC {  
    public static void main(String[] args) {  
        StringBuffer s1 = new StringBuffer("Ciao");  
        StringBuffer s2 = new StringBuffer("Addio");  
        System.out.println(s1);  
        // l'oggetto riferito da s1 non e' ancora  
        // eleggibile per GC  
        s1 = s2;  
        // ora e' eleggibile.  
    }  
}
```

Ma questo non è l'ultimo modo di rendere eleggibile un oggetto!

LP1 – Lezione 14

12 / 18

Ancora???

```
1. import java.util.Date;  
2. public class TestGC {  
3.     public static void main(String[] args) {  
4.         Date d = getDate();  
5.         System.out.println(d);  
6.     }  
7.  
8.     public static Date getDate() {  
9.         Date d2 = new Date();  
10.        String now = d2.toString();  
11.        System.out.println(now);  
12.        return d2;  
13.    }  
14. }
```

Tracciare il codice con riferimento alla eleggibilità per GC degli oggetti.

Ma questo non è l'ultimo modo di rendere eleggibile un oggetto!

LP1 – Lezione 14

13 / 18

Mo' basta!

```
public class Isola {  
    Isola i;  
  
    public static void main(String[] args) {  
        Isola i1 = new Isola();  
        Isola i2 = new Isola();  
        Isola i3 = new Isola();  
  
        i1.i = i2;  
        i2.i = i3;  
        i3.i = i1;  
  
        i1 = null;  
        i2 = null;  
        i3 = null;  
  
        // qui quali oggetti sono eleggibili?  
    }  
}
```

LP1 – Lezione 14

14 / 18

D 1

Data la stringa costruita mediante `s = new String("xyzzy")`, quali delle seguenti invocazioni di metodi modificherà la stringa?

- A. `s.append("aaa");`
- B. `s.trim();`
- C. `s.substring(3);`
- D. `s.replace('z', 'a');`
- E. `s.concat(s);`
- F. Nessuna delle precedenti.

LP1 – Lezione 14

16 / 18

D 2

Qual è l'output del seguente brano di codice?

```
1. String s1 = "abc" + "def";
2. String s2 = new String(s1);
3. if (s1 == s2)
4.     System.out.println("A");
5. if (s1.equals(s2))
6.     System.out.println("B");
```

- A. AB
- B. A
- C. B
- D. Nessun output.

LP1 – Lezione 14

17 / 18

D 3

Quanti oggetti sono prodotti nel seguente frammento di codice?

```
1. StringBuffer sbuf = new StringBuffer("abcde");
2. sbuf.insert(3, "xyz");
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

LP1 – Lezione 14

18 / 18

Linguaggi di Programmazione I – Lezione 5

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

1 aprile 2008

Diagrammi UML	3
UML: richiami	4
Modello statico	5
- Use Case	6
- Class	7
- Object	8
- Component	9
- Deployment	10
Modello dinamico	11
- Sequence	12
- Collaboration	13
- State	14
- Activity	15
Fase di analisi iniziale	16
Attività	17
Reperire info	18
Evitare pregiudizi	19
Esperti del dominio	20
Enunciato del	21
Dominio del	22
Possibili oggetti	23
Esempio	24
Dizionario dati	25
Esempio (cont.)	26
Esercizio/Progetto	27
Creare gli Use	28
Esempio (cont.)	29
Esempio (cont.)	30
Scenari di	31
Esempio (cont.)	32
Esercizio/Progetto	33
Diagrammi Activity	34
Esercizio/Progetto	35
Bibliografia	36

Bibliografia.	36
-----------------------	----

Panoramica della lezione

Diagrammi UML

Fase di analisi iniziale

Bibliografia

LP1 – Lezione 5

2 / 36

Diagrammi UML

3 / 36

UML: richiami

È un linguaggio grafico per:

- Specificare
- Costruire
- Visualizzare
- Documentare

È la notazione che descrive il ciclo completo dello sviluppo OO.

Diverse scritture degli stessi diagrammi, con maggiore o minore dettaglio, a seconda dell'ambito di lettura e della prospettiva scelta per visualizzare il sistema.

LP1 – Lezione 5

4 / 36

Modello statico

- Rappresenta la struttura base del sistema software che si sta modellando.
- Costruisce e documenta l'aspetto statico del sistema.
- Crea una rappresentazione degli elementi principali del dominio del problema.
- Comprende i diagrammi:

Use Case: utili per identificare le proprietà di base o le specifiche richiesta al sistema.

Class: sono i più usati; essi forniscono diverse visualizzazioni del sistema, sono uno strumento per esplorare il vocabolario del dominio del problema, sono un modo per documentare successive realizzazioni.

Object: aiutano ad identificare l'organizzazione strutturale tra oggetti.

Component: aiutano a localizzare applicazioni software.

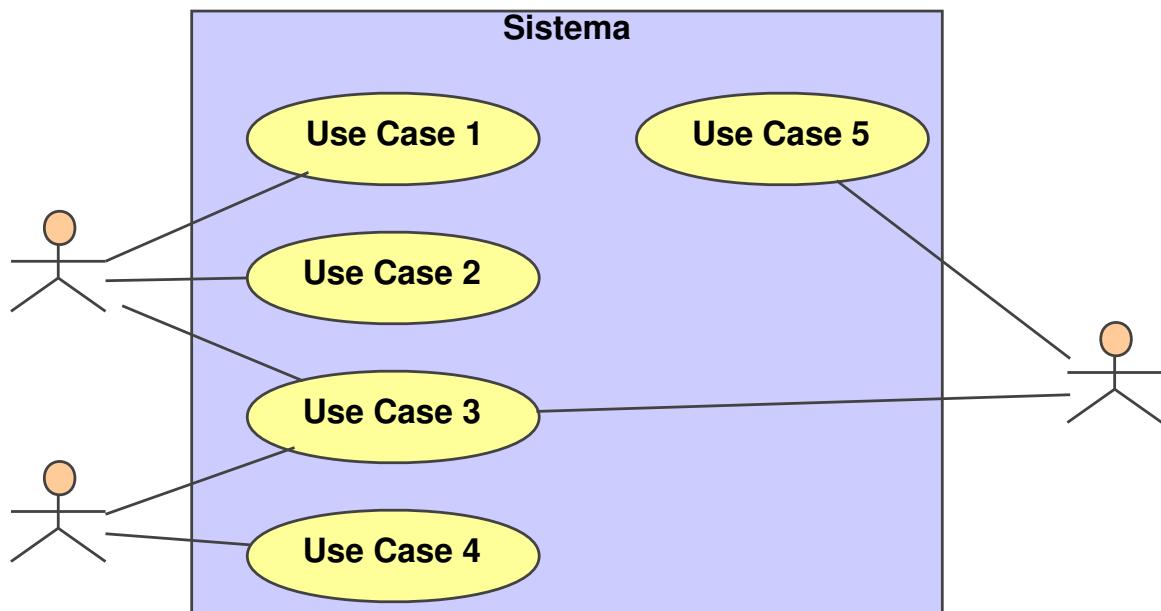
Deployment: specificano la mappa hardware.

LP1 – Lezione 5

5 / 36

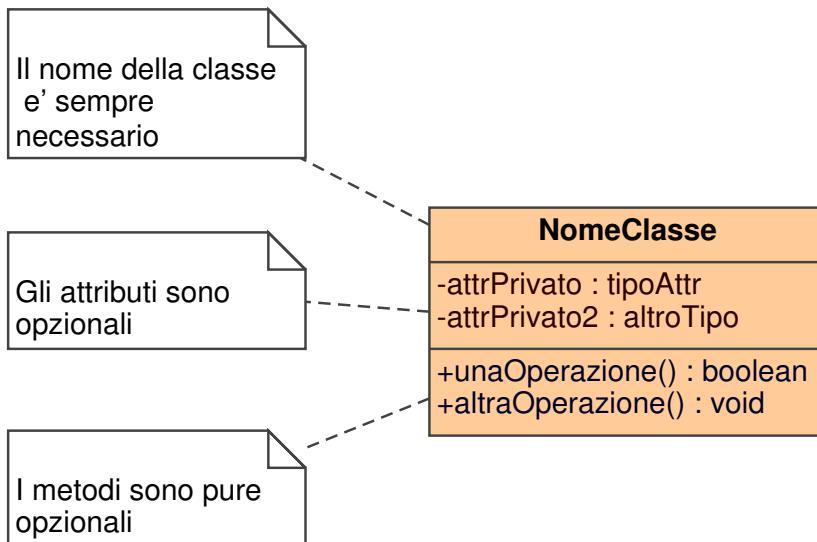
- Use Case

- Mette in evidenza le specifiche del sistema.
- Mostra chi o cosa usa il sistema.
- Gli utenti di una certa funzionalità sono detti *attori*.
- I casi d'uso sono rappresentati con ellissi.



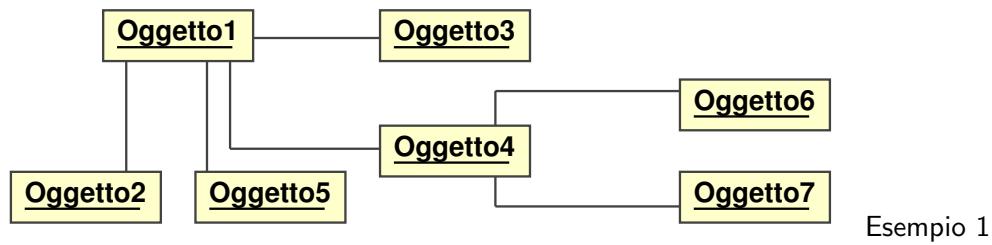
- Class

- Diagrammi concettuali: usati spesso dagli analisti per meglio individuare i concetti ed il vocabolario del dominio del problema. In questa prospettiva bisogna escludere i dettagli realizzativi specifici del linguaggio.
- Diagrammi realizzativi: contenenti maggiori dettagli riguardanti metodi ed attributi necessari per realizzare le classi.



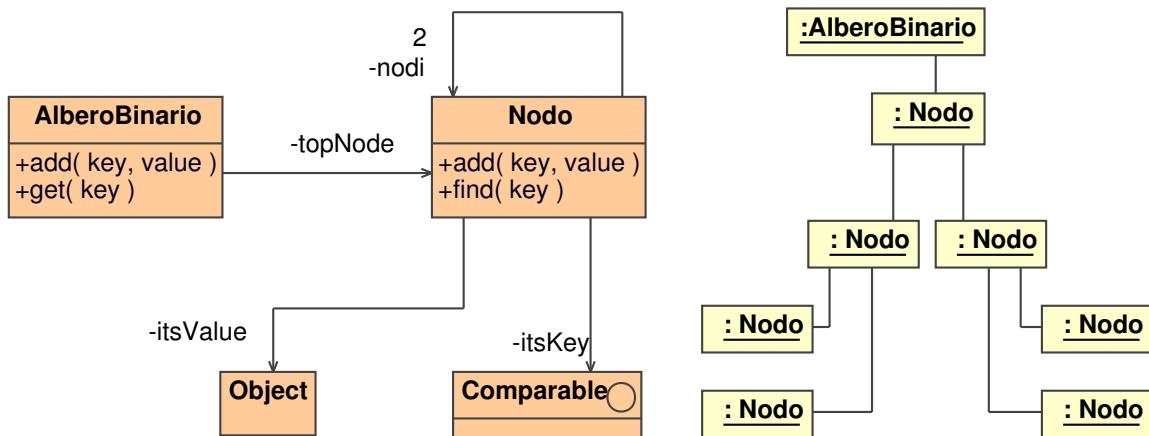
- Object

È un diagramma usato per rappresentare oggetti specifici che esistono ad un certo istante durante il ciclo di vita del sistema.

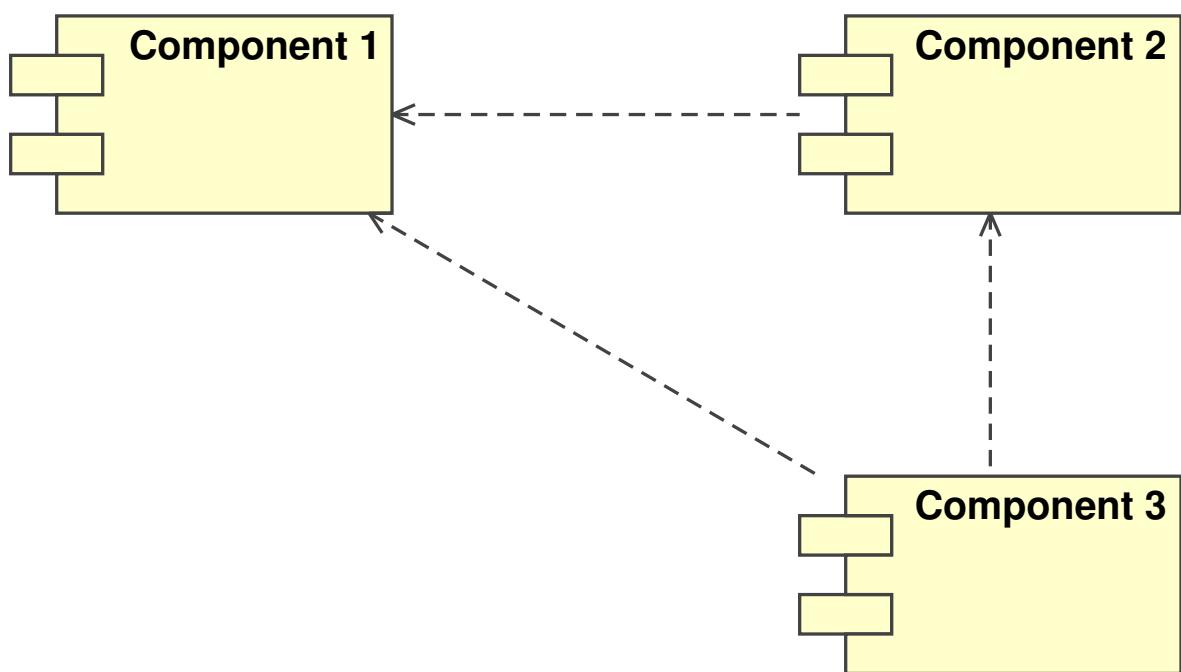


Esempio 1

Esempio 2 (dalle classi agli oggetti):



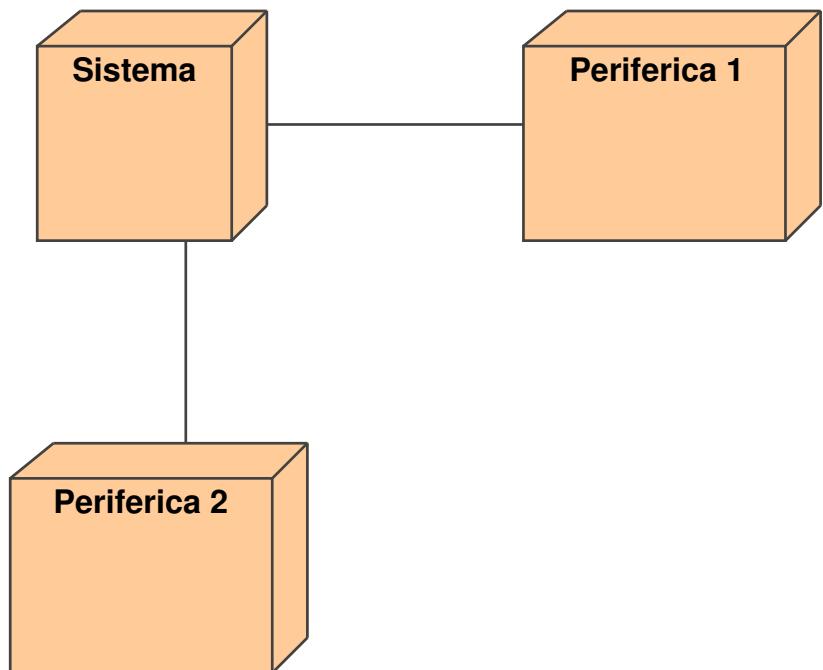
- Component



- Mostrano le relazioni tra componenti software.
- Le dipendenze sono rappresentate da linee tratteggiate.

- Deployment

Rappresenta i dispositivi fisici che saranno usati per eseguire le applicazioni software.



Modello dinamico

- Riguarda il comportamento del sistema.
- Comprende i diagrammi:

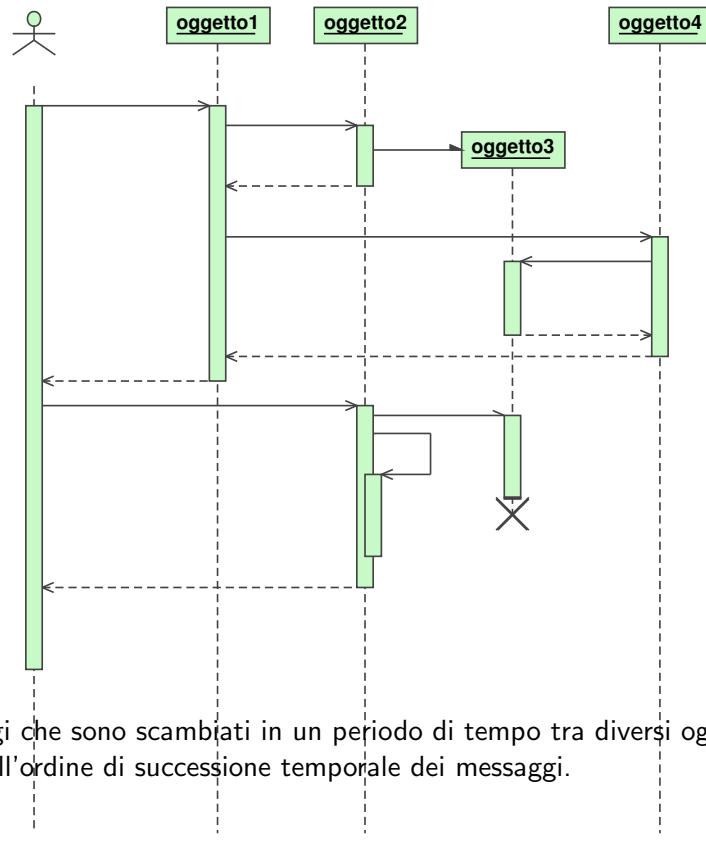
Sequence: mostrano la sequenza temporale di messaggi durante una particolare attività del sistema.

Collaboration: mostrano messaggi tra oggetti e la struttura tra gli oggetti.

State: esaminano gli stati interni e le transizioni tra stati di un singolo oggetto.

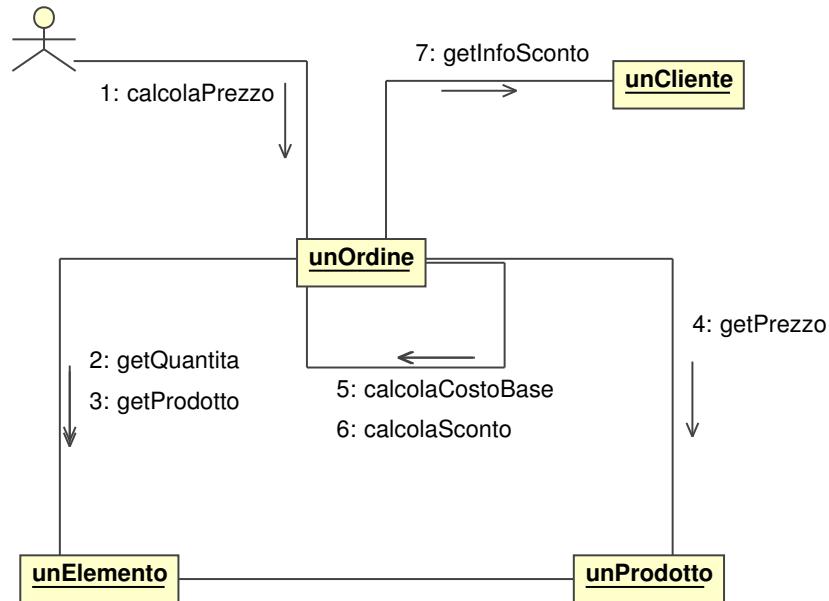
Activity: esaminano il comportamento di più oggetti.

- Sequence



- Cattura i messaggi che sono scambiati in un periodo di tempo tra diversi oggetti.
- Pone l'accento sull'ordine di successione temporale dei messaggi.

- Collaboration

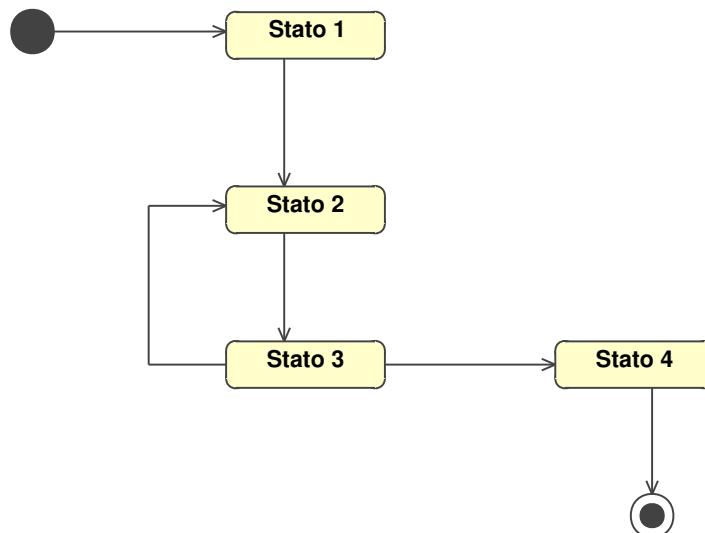


- Mostra la collaborazione di oggetti mediante messaggi.
- I messaggi possono essere numerati a vari livelli.
- Pone l'accento sull'organizzazione strutturale degli oggetti che scambiano messaggi.

LP1 – Lezione 5

13 / 36

- State



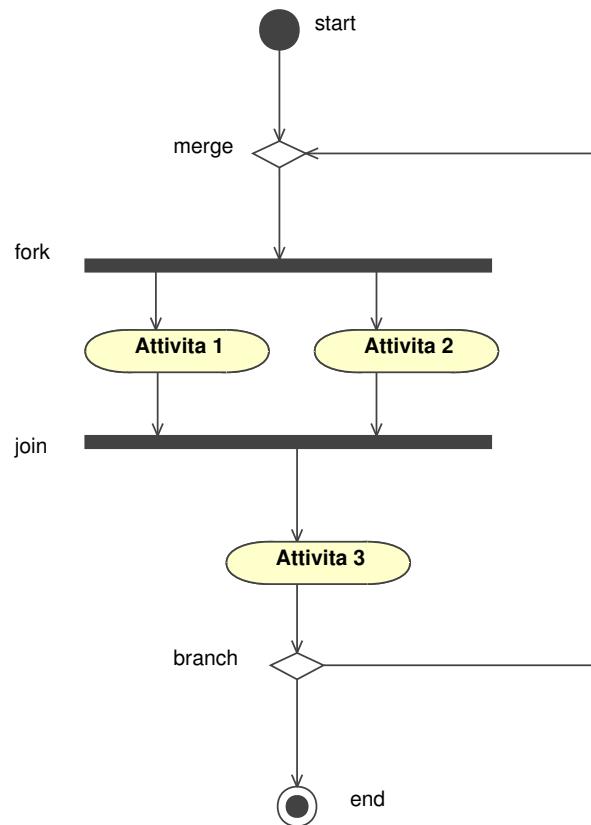
- Mostra una rappresentazione di un singolo oggetto come automa a stati finiti.
- Pone l'accento sul comportamento di un singolo oggetto in risposta ad eventi esterni.

LP1 – Lezione 5

14 / 36

- Activity

Descrivono il flusso da una attività alla successiva.



Fase di analisi iniziale

Attività

Durante la fase di analisi iniziale:

- viene individuato il problema;
- sono analizzati i flussi di lavoro di ciascuna iterazione nel ciclo di sviluppo;
- sono raccolte le informazioni;
- viene creato l'enunciato del problema;
- sono generati i diagrammi Use Case;

Reperire info

Si possono reperire informazioni attraverso questionari o gruppi di discussione da numerose fonti:

- Richieste iniziali del cliente: di solito un documento non sempre preciso; bisogna intervistare le persone giuste, riempire vuoti, capire in anticipo la tecnologia necessaria.
- Esperti del dominio: nel settore di attività del cliente, di solito analisti presentati dal cliente.
- Utenti finali: sono coloro che forniranno informazioni pratiche sull'uso del prodotto, contro le informazioni teoriche precedenti.
- Dirigenti: sono coloro che dovranno gestire il prodotto.
- Mercato: deve essere esplorato per riconoscere se esiste una richiesta per prodotti simili; se è così si potrebbe considerare di costruire un modulo o componenti generiche per promuovere il riuso in progetti simili;
- Precedenti progetti: forniscono informazioni (positive o negative) su altre esperienze.

LP1 – Lezione 5

18 / 36

Evitare pregiudizi

Per esempio:

- Gli utenti sono ingenui, gli sviluppatori no.
- Le richieste o le specifiche sono statiche.
- Si può costruire la soluzione corretta al primo tentativo.

Per evitarli:

- ricordare che il sistema deve fornire funzionalità richieste: i progetti evolvono costantemente e le richieste cambiano;
- riconoscere che non si può produrre un prodotto perfetto nella prima fase;
- identificare chiaramente le richieste dell'utente;
- assicurarsi che il modello possa adattarsi a richieste in evoluzione;
- prevedere di poter correggere il modello se vi saranno imprecisioni di informazioni nel dominio del problema.

LP1 – Lezione 5

19 / 36

Esperti del dominio

Sono gli specialisti in un'area particolare relativa al problema.

Possono essere divisi rozzamente nelle seguenti categorie:

- Esperti generali del dominio.
- Esperti di specifiche applicazioni nel dominio.
- Esperti di domini simili a quello in oggetto.

LP1 – Lezione 5

20 / 36

Enunciato del problema

È il documento che descrive chiaramente le richieste del cliente. Esso deve dichiarare:

- tutte le informazioni che sono pertinenti all'analisi e al disegno del progetto;
- tutti i vincoli esistenti che il progetto di sviluppo del sistema deve tenere in considerazione;
- i flussi di informazione attraverso il sistema;
- gli utenti finali del prodotto;
- l'ingresso e l'uscita del sistema.

Deve essere scritto in un linguaggio specifico del dominio del problema. Deve contenere frasi complete, senza abbreviazioni, e nessuna o poca terminologia di informatica.

Può evolvere.

È usato come base per individuare il dominio del problema.

LP1 – Lezione 5

21 / 36

Dominio del problema

È l'enunciato, grafico o in forma di testo, che descrive quali aree e problemi dovranno essere di pertinenza del sistema.

Il cliente non è coinvolto a questo livello. Dovrà essere coinvolto durante il progetto, per chiarire e convalidare tutte le aree del dominio del problema.

LP1 – Lezione 5

22 / 36

Possibili oggetti e classi

È necessario identificare ora una lista di oggetti e classi che il sistema deve contenere.

- Identificate dall'enunciato del problema.
- Sottolineare sostantivi dall'enunciato del problema per costruire la lista degli oggetti e delle classi candidate.

LP1 – Lezione 5

23 / 36

Esempio

Il Jolly Hotel di Napoli richiede un pacchetto software per facilitare l'automazione di molti compiti manuali già svolti dallo staff dell'hotel.

L'hotel contiene un certo numero di camere. Le informazioni rilevanti per ciascuna camera sono:

- numero della camera
- prezzo base
- numero massimo di ospiti
- tipo (es: singola, doppia, doppia con uso singola, executive, suite).

Il costo della camera è il prezzo base più altri costi stagionali.

Un ospite potenziale può riservare per telefono una o più camere per un periodo specifico.

Queste prenotazioni sono ricevute da un operatore telefonico.

L'ospite potenziale richiede un tipo di camera, una data di arrivo, una data di partenza.

Venne effettuata una ricerca per la disponibilità della camera per le date richieste. Se

disponibile, il cliente è informato sui dettagli e sul costo.

Se il preventivo è accettato, viene formulata una prenotazione provvisoria. Questa prenotazione è valida per una durata di tempo stabilita dall'operatore telefonico.

La prenotazione provvisoria diventa prenotazione definitiva quando viene ricevuto e confermato il pagamento di una cauzione.

Esso può avvenire nel momento della prenotazione iniziale.

Il receptionist può anche accettare una prenotazione per ospiti potenziali che arrivano in hotel senza una prenotazione. In questo caso non è richiesto il pagamento della cauzione, ma è sufficiente l'identificazione dell'ospite.

Venne annotato quando l'ospite prende possesso della camera (check in) e quando la rilascia (check out).

Il telefono interno è abilitato/disabilitato durante il check in/out, attraverso centralino.

Il centralino registra anche durata e costo delle telefonate uscenti da ciascuna camera.

LP1 – Lezione 5

24 / 36

Dizionario dati

- È il documento che descrive tutto il vocabolario usato per il progetto.
- È particolarmente importante per grandi progetti in cui sono coinvolti numerosi gruppi di lavoro.
- I termini durano per tutto il progetto.
- Si possono introdurre nuovi termini.
- Aiuta ad eliminare ambiguità.
- Deve essere facilmente reperibile da tutti i membri del gruppo di progetto.
- Deve essere costantemente aggiornato.

LP1 – Lezione 5

25 / 36

Esempio (cont.)

Numero camera: Un numero che identifica univocamente la camera nell'hotel. La cifra iniziale indica il piano. Il range è da 1 a 750.

Prezzo base: Il prezzo di riferimento senza servizi aggiuntivi o offerte speciali.

Massima occupabilità: Ciascuna camera può ospitare in modo sicuro e confortevole un particolare numero di persone.

Tipo camera: Un tipo di camera; per esempio: singola, doppia, doppia con uso singola, suite. Il tipo di camera dipende dalla dimensione, posizione, arredamento e accessori aggiuntivi.

Check in: Quando l'ospite arriva all'hotel e richiede la camera che aveva prima riservato.

Check out: Quando l'ospite lascia l'hotel dopo aver pagato il conto.

Receptionist: Un membro dello staff dell'hotel, responsabile del check in/out degli ospiti e della registrazione per ospiti potenziali che arrivano senza prenotazione.

Segretario: Un membro dello staff dell'hotel, responsabile delle prenotazioni.

Prenotazione provvisoria: L'inizializzazione di una richiesta per una particolare camera, per un particolare periodo di tempo. Le camere sono bloccate per breve tempo, in attesa della conferma. Se non arriva in tempo una conferma la camera è sbloccata e resa nuovamente disponibile per la prenotazione.

LP1 – Lezione 5

26 / 36

Esercizio/Progetto (Prima parte)

1. Scrivere l'enunciato del problema per un sistema a scelta libera.
Inizialmente esso è delineato da persone non esperte dell'argomento.
2. Chi sono gli esperti del dominio per tale problema?
3. Sottolineare i sostantivi presenti nell'enunciato del problema e produrre una lista di oggetti candidati.
4. Preparare un dizionario di dati per il sistema.

LP1 – Lezione 5

27 / 36

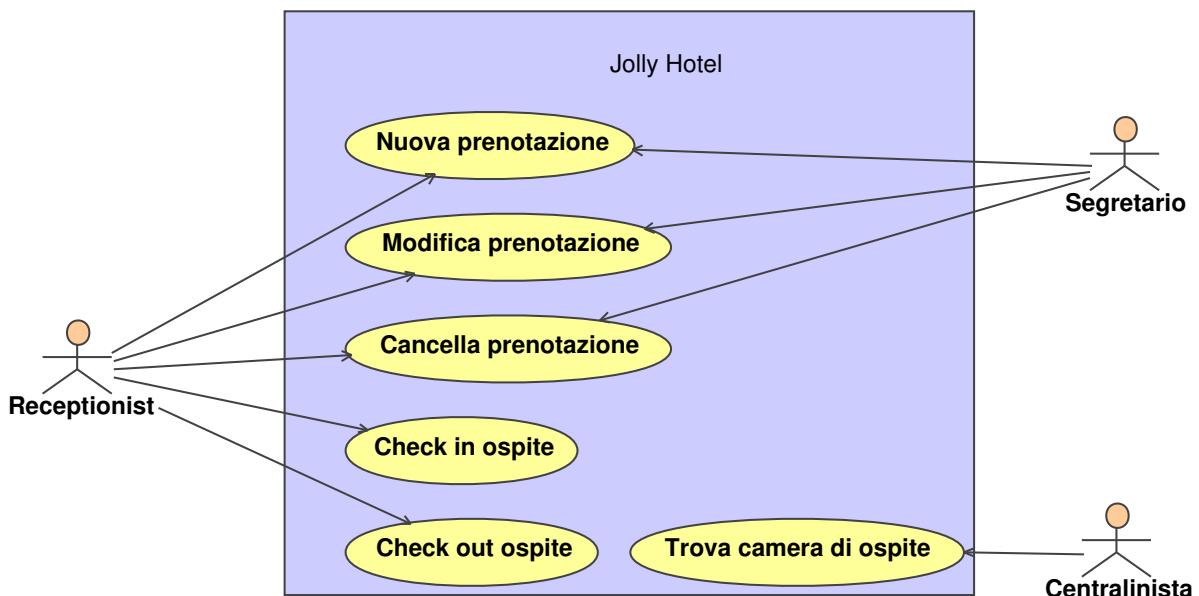
Creare gli Use Case

- Uno Use Case è una rappresentazione grafica e schematica delle interazioni dell'utente con il sistema.
- Assiste nella determinazione delle specifiche del sistema e del suo contesto operativo.
- È rappresentato graficamente mediante una ellisse.
- Un *diagramma* di Use Case è composto da vari Use Case. Gli Attori sono associati agli Use Case quando avviene uno scambio di comunicazioni.

LP1 – Lezione 5

28 / 36

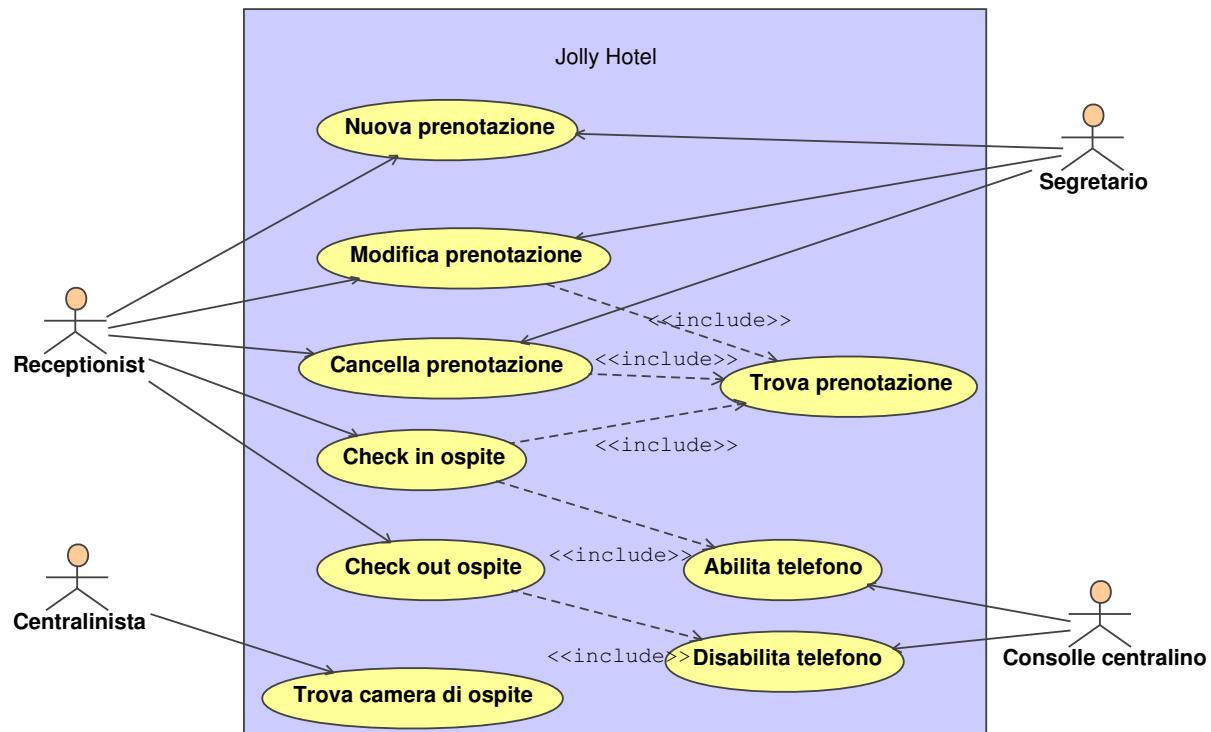
Esempio (cont.)



LP1 – Lezione 5

29 / 36

Esempio (cont.)



LP1 – Lezione 5

30 / 36

Scenari di Use Case

- Uno Use Case descrive una funzione fondamentale del sistema dal punto di vista dell'utente.
- Uno scenario fa riferimento ad una istanza di uno Use Case, cioè un percorso logico dall'inizio alla fine.
- Gli scenari non contengono enunciati condizionali.
- Cominciano nello stesso modo, possono finire in modo diverso.
- Scrivere solo gli scenari principali.
- Includere sia quelli che si concludono con successo, sia quelli con insuccesso.

LP1 – Lezione 5

31 / 36

Esempio (cont.)

Scenari dello Use Case: *Nuova prenotazione*:

- **Scenario 1:** Al Segretario viene presentato un modulo in cui viene richiesto il tipo di camera, la data di arrivo, la durata del soggiorno, la data di partenza.
Il cliente richiede una stanza doppia dal 31 gennaio al 5 febbraio. L'informazione viene inoltrata e parte la ricerca. Il sistema risponde che la stanza è disponibile e mostra il prezzo. Dopo la conferma da parte del cliente, il segretario accetta la prenotazione, che viene etichettata come provvisoria. Un codice di prenotazione viene comunicato al cliente. Esso può confermare la prenotazione pagando un anticipo.
- **Scenario 2:** Al Segretario è presentato un modulo in cui viene richiesto il tipo di camera, una data di arrivo, la

durata del soggiorno, la data di partenza.

Il cliente richiede una stanza singola, dal 3 dicembre al 14 dicembre. L'informazione viene inoltrata e parte la ricerca. Il sistema risponde che la stanza non è disponibile. Viene chiesto al cliente di scegliere un diverso tipo di stanza o altre date per il soggiorno.

- **Scenario 3:** Al Segretario è presentato un modulo in cui viene richiesto il tipo di camera, una data di arrivo, la durata del soggiorno, la data di partenza.
Il cliente richiede una suite, dal 20 giugno al 2 luglio. L'informazione viene inoltrata e parte la ricerca. Il sistema risponde che la stanza è disponibile e mostra il prezzo. Il cliente non accetta l'offerta.

LP1 – Lezione 5

32 / 36

Esercizio/Progetto (Seconda parte)

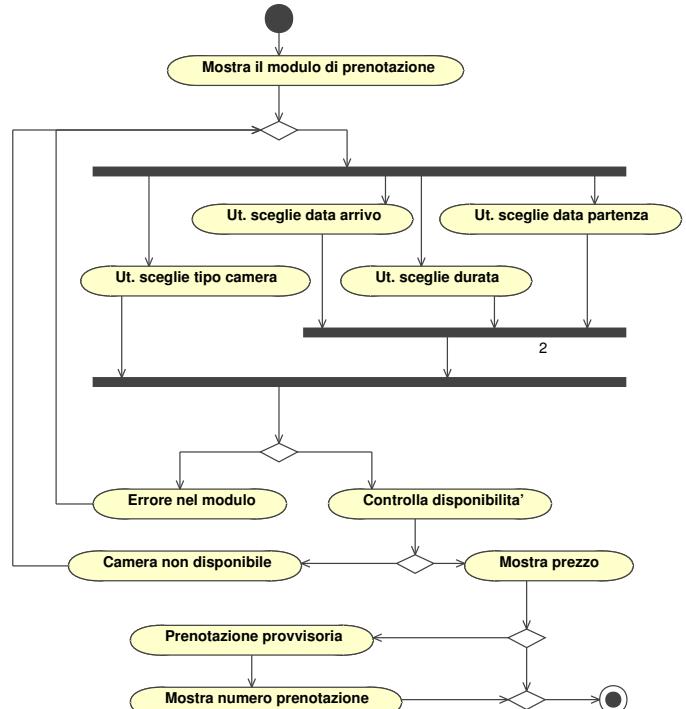
1. Analizzare l'enunciato del problema scelto e produrre un diagramma Use Case.
2. Preparare alcuni scenari per uno degli Use Case del diagramma.

LP1 – Lezione 5

33 / 36

Diagrammi Activity

- Mostrano attività, processi, o flussi.
- Usati dovunque sia necessario costruire un modello di alcune attività.
- Possono essere usati, per esempio, per fornire una visualizzazione grafica a complemento degli scenari di uno Use Case.



LP1 – Lezione 5

34 / 36

Esercizio/Progetto (Terza parte)

1. Produrre un diagramma Activity relativo allo Use Case scelto per produrre gli scenari dell'esercizio precedente.

LP1 – Lezione 5

35 / 36

Bibliografia

36 / 36

Bibliografia

- Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language Reference Manual*. Addison-Wesley.
- Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*. Addison-Wesley.
- Fowler M. *UML distilled*. Addison-Wesley.

LP1 – Lezione 5

36 / 36

Linguaggi di Programmazione I – Lezione 15

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

5 giugno 2008

Eccezioni: il meccanismo	3
Introduzione	4
try e catch	5
finally	6
Vincoli.	7
Propagazione	8
Eccezioni: i dettagli	9
Definizione	10
Gerarchia (1)	11
Gerarchia (2)	12
Eccez. catturate (1)	13
Eccez. catturate (2)	14
Eccez. catturate (3)	15
... e non catturate	16
<i>Handle or Declare</i>	17
Esempio (1)	18
Esempio (2)	19
Esempio (3)	20
Esempio (4)	21
Nuove eccezioni	22
Overriding (ancora?)	23
Esercizi	24
Esercizi	24
Questionario	25
D 1	26
D 2	27
D 3	28
D 4	29
D 5	30
D 6	31
D 7	32
D 8	33
D 9	34

D 10	35
D 11	36
D 12	37
D 13	38

Eccezioni (Gestione degli errori)

Eccezioni: il meccanismo

Eccezioni: i dettagli

Esercizi

Questionario

LP1 – Lezione 15

2 / 38

Eccezioni: il meccanismo

3 / 38

Introduzione

- Le eccezioni denotano “eventi eccezionali” la cui occorrenza altera il flusso normale delle istruzioni.
- Es.: risorse hardware indisponibili, hardware malfunzionante, bachi nel software ...
- Quando capita un tale evento, si dice che viene “lanciata una eccezione”.

LP1 – Lezione 15

4 / 38

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.

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

LP1 – Lezione 15

5 / 38

finally

- Un blocco opzionale marcato `finally` verrà (quasi) SEMPRE eseguito, anche dopo il lancio ed la manipolazione (eventuale) dell'eccezione.

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

- Il blocco `finally` viene eseguito perfino dopo una eventuale istruzione `return` presente nei blocchi `try` o `catch`.
- IL BLOCCO `finally` VIENE ESEGUITO SEMPRE.
- Il blocco `finally` potrebbe non essere eseguito o potrebbe non completare l'esecuzione solo in conseguenza di un crash totale del sistema oppure tramite una invocazione di `System.exit(int status)`.

LP1 – Lezione 15

6 / 38

Vincoli

- Le clausole `catch` ed 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` (vedremo tra poco).

LP1 – Lezione 15

7 / 38

Propagazione

- Perché le clausole `catch` non sono obbligatorie?
- Che succede ad una eccezione che viene lanciata da un blocco `try`, ma per la quale non esiste una clausola `catch` che l'attende?
- Una eccezione non catturata semplicemente “si immerge” nel record di attivazione che la ha generata, “riemergendo” nel successivo record di attivazione.
- Qui potrebbe essere catturata da una ulteriore clausola `catch`, oppure altrimenti ricadere nel record successivo, e così via. Finché, 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).

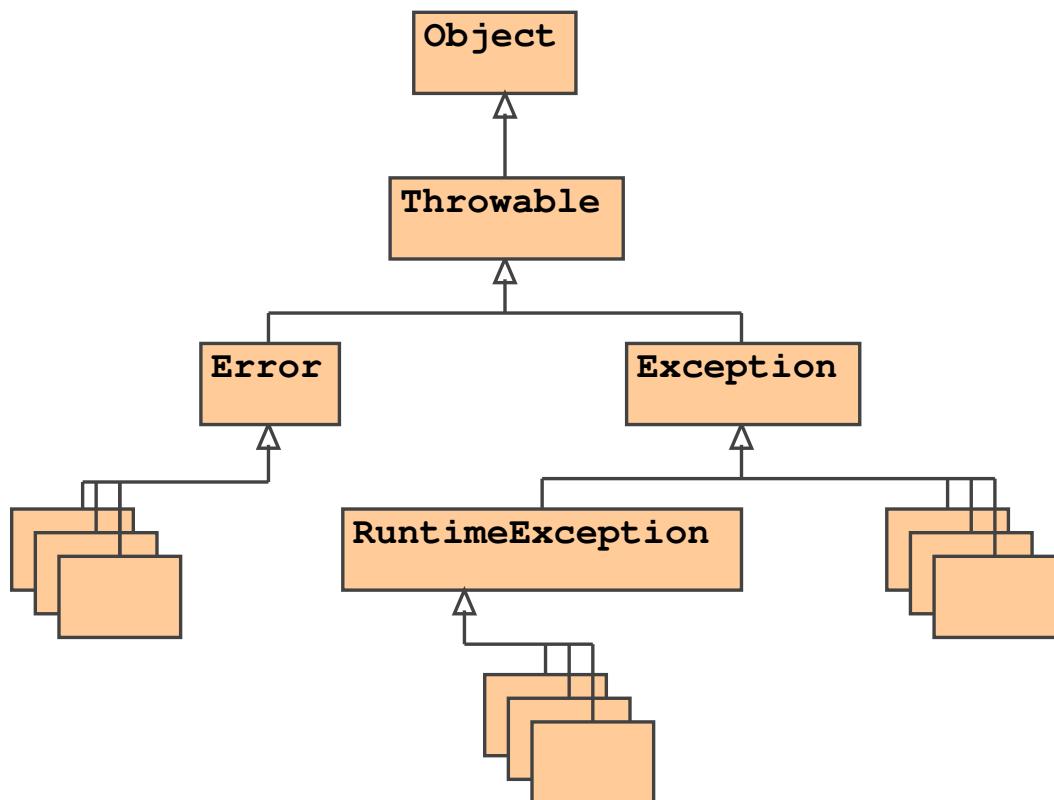
LP1 – Lezione 15

8 / 38

Definizione

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno “eccezione” a questa regola.
 - Ogni eccezione è una istanza di una sotto-classe 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`).
  - La gerarchia degli oggetti lanciabili è la seguente:

## Gerarchia (1)



## Gerarchia (2)

- 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. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile. In genere, una applicazione non deve essere capace di riprendersi da una situazione di errore. 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 (errori di programmazione, bachi). Sono qui evidenziati perché indicano eccezioni rare e difficili da gestire (discuteremo in seguito). Il programmatore, che si accorge di un baco dovuto ad un suo errore, deve correggerlo, non gestirlo come una eccezione!

LP1 – Lezione 15

12 / 38

## Eccezioni catturate (1)

- Una clausola `catch` cattura ogni oggetto-eccezione il cui tipo può essere ricondotto mediante conversione automatica al tipo specificato nella clausola.
- Esempio: la classe `IndexOutOfBoundsException` ha due sottoclassi, `ArrayIndexOutOfBoundsException` e `StringIndexOutOfBoundsException`; si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:

```
try {
 // Codice che potrebbe lanciare una eccezione
 // IndexOutOfBoundsException oppure
 // ArrayIndexOutOfBoundsException oppure
 // StringIndexOutOfBoundsException
}
catch (IndexOutOfBoundsException e) {
 e.printStackTrace();
}
```

LP1 – Lezione 15

13 / 38

## Eccezioni catturate (2)

- Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {
 // codice rischioso
}
catch (Exception e) {
}
```

- L'ordine delle clausole catch è importante.
- Nell'esempio precedente, se avessimo scritto:

```
try {
 // Codice che potrebbe lanciare una eccezione
 // IndexOutOfBoundsException oppure
 // ArrayIndexOutOfBoundsException
}
catch (IndexOutOfBoundsException e) {
 // tratta l'eccezione
}
catch (ArrayIndexOutOfBoundsException e) {
 // tratta l'eccezione
}
```

il codice non sarebbe stato compilato.

## Eccezioni catturate (3)

- È corretto, invece, scrivere:

```
try {
 // Codice che potrebbe lanciare una eccezione
 // IndexOutOfBoundsException oppure
 // ArrayIndexOutOfBoundsException
}
catch (ArrayIndexOutOfBoundsException e) {
 // tratta l'eccezione
}
catch (IndexOutOfBoundsException e) {
 // tratta l'eccezione
}
```

### ... e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di `RuntimeException`).
- La parola chiave `throws` viene usata per elencare le eccezioni che possono fuoriuscire da un metodo:

```
void miaFunzione() throws MiaEccezione1, MiaEccezione2 {
 // qui il codice per il metodo
}
```
- Il fatto che un metodo dichiara l'eccezione non significa che esso la lancerà sempre, ma avverte l'utilizzatore che esso potrebbe lanciarla.

LP1 – Lezione 15

16 / 38

### 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 *handle or declare*):
  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`.
  2. Propagare l'eccezione dichiarandola nell'intestazione del metodo.
- Una “eccezione” a questa regola: le `RuntimeException` sono esenti dall'obbligo di dichiarazione. Esse sono *unchecked* dal compilatore, mentre le rimanenti eccezioni sono dette *checked*. Ora, forse, si capisce il motivo della gerarchia precedentemente esposta.
- Nota: l'*or* della regola non è esclusivo, nel senso che si può decidere di fare entrambe le cose.

LP1 – Lezione 15

17 / 38

### Esempio (1)

Quali problemi ci sono in questo codice?

```
void f1() {
 f2();
}

void f2() {
 throw new IOException();
}
```

- Il metodo `f2` lancia una eccezione *checked* ma non la dichiara;
- Se esso l'avesse dichiarata, come in:

```
void f2() throws IOException {...}
```

il problema l'avrebbe `f1` che dovrebbe ora dichiararla o catturarla.

LP1 – Lezione 15

18 / 38

## Esempio (2)

Quali problemi ci sono in questo codice?

```
import java.io.*;
class Test {
 public int f1() throws EOFException {
 return f2();
 }
 public int f2() throws EOFException {
 // qui il codice che lancia effettivamente l'eccezione
 return 1;
 }
}
```

- Poiché EOFException è sottoclasse di IOException, che è sottoclasse di Exception, essa è una eccezione *checked*. Essa viene regolarmente dichiarata ed il codice regolarmente compilato.

LP1 – Lezione 15

19 / 38

## Esempio (3)

Quali problemi ci sono in questo codice?

```
public void f1() {
 // qui codice che puo' lanciare NullPointerException
}
```

- Poiché NullPointerException è sottoclasse di RuntimeException, essa è una eccezione *unchecked*. Non è necessario né dichiararla, né catturarla, ed il codice viene regolarmente compilato.

LP1 – Lezione 15

20 / 38

## Esempio (4)

Analogamente, questo codice compila correttamente:

```
class TestEx {
 public static void main (String [] args) {
 mioMetodo();
 }
 static void mioMetodo() { // Non c'e' bisogno di
 // dichiarare un Error
 fai();
 }
 static void fai() { // Non c'e' bisogno di dichiarare un Error
 try {
 throw new Error();
 }
 catch(Error me) {
 throw me; // Ti ho preso, ma ora di rilancio
 }
 }
}
```

LP1 – Lezione 15

21 / 38

## 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`.
- Pertanto, il codice seguente non compila:

```
class TestEx {
 void f() {
 throw new MiaEccezione();
 }
}
```

## Overriding (ancora?)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per la sovrapposizione di metodi**:

- I due metodi devono avere identica segnatura.
- I due 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 metodo nella sottoclasse, DEVONO essere tipi posti al di sotto nella gerarchia delle eccezioni dichiarate dal metodo nella superclasse.

## Esercizi

### Esercizi

Sono proposti in allegato due esercizi:

1. In questo esercizio si dovranno usare i blocchi `try-catch` per gestire una semplice `RuntimeException`.
2. In questo esercizio si dovrà creare il nuovo tipo `OverdraftException` di oggetti lanciabili dal metodo `preleva` nella classe `Conto`.

**D 1**

Dato il codice seguente:

```

1. try {
2. // codice rischioso; hp:
3. // Exception
4. // +--- EccA
5. // +--- EccB
6. // +--- EccC
7. System.out.print(1);
8. }
9. catch (EccB e) {
10. System.out.println(2);
11. }
12. catch (EccC e) {
13. System.out.println(3);
14. }
15. catch (Exception e) {
16. System.out.println(4);
17. }
18. finally {

```

```

19. System.out.println(5);
20. }
21. System.out.println(6);

```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccB?

- A.** 1
- B.** 2
- C.** 3
- D.** 4
- E.** 5
- F.** 6

LP1 – Lezione 15

26 / 38

**D 2**

Dato il codice seguente:

```

1. try {
2. // codice rischioso; hp:
3. // Exception
4. // +--- EccA
5. // +--- EccB
6. // +--- EccC
7. System.out.print(1);
8. }
9. catch (EccB e) {
10. System.out.println(2);
11. }
12. catch (EccC e) {
13. System.out.println(3);
14. }
15. catch (Exception e) {
16. System.out.println(4);
17. }
18. finally {

```

```

19. System.out.println(5);
20. }
21. System.out.println(6);

```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 non venga lanciata alcuna eccezione?

- A.** 1
- B.** 2
- C.** 3
- D.** 4
- E.** 5
- F.** 6

LP1 – Lezione 15

27 / 38

## D 3

Dato il codice seguente:

```
1. try {
2. // codice rischioso; hp:
3. // Exception
4. // +--- EccA
5. // +--- EccB
6. // +--- EccC
7. System.out.print(1);
8. }
9. catch (EccB e) {
10. System.out.println(2);
11. }
12. catch (EccC e) {
13. System.out.println(3);
14. }
15. catch (Exception e) {
16. System.out.println(4);
17. }
18. finally {
```

```
19. System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccC?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

LP1 – Lezione 15

28 / 38

## D 4

Dato il codice seguente:

```
1. try {
2. // codice rischioso; hp:
3. // Exception
4. // +--- EccA
5. // +--- EccB
6. // +--- EccC
7. System.out.print(1);
8. }
9. catch (EccB e) {
10. System.out.println(2);
11. }
12. catch (EccC e) {
13. System.out.println(3);
14. }
15. catch (Exception e) {
16. System.out.println(4);
17. }
18. finally {
```

```
19. System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccA?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

LP1 – Lezione 15

29 / 38

## D 5

Dato il codice seguente:

```
1. try {
2. // codice rischioso; hp:
3. // Exception
4. // +--- EccA
5. // +--- EccB
6. // +--- EccC
7. System.out.print(1);
8. }
9. catch (EccB e) {
10. System.out.println(2);
11. }
12. catch (EccC e) {
13. System.out.println(3);
14. }
15. catch (Exception e) {
16. System.out.println(4);
17. }
18. finally {
```

```
19. System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Exception?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

LP1 – Lezione 15

30 / 38

## D 6

Dato il codice seguente:

```
1. try {
2. // codice rischioso; hp:
3. // Exception
4. // +--- EccA
5. // +--- EccB
6. // +--- EccC
7. System.out.print(1);
8. }
9. catch (EccB e) {
10. System.out.println(2);
11. }
12. catch (EccC e) {
13. System.out.println(3);
14. }
15. catch (Exception e) {
16. System.out.println(4);
17. }
18. finally {
```

```
19. System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo RuntimeException?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

LP1 – Lezione 15

31 / 38

## D 7

Dato il codice seguente:

```
1. try {
2. // codice rischioso; hp:
3. // Exception
4. // +--- EccA
5. // +--- EccB
6. // +--- EccC
7. System.out.print(1);
8. }
9. catch (EccB e) {
10. System.out.println(2);
11. }
12. catch (EccC e) {
13. System.out.println(3);
14. }
15. catch (Exception e) {
16. System.out.println(4);
17. }
18. finally {
```

```
19. System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Error?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

LP1 – Lezione 15

32 / 38

## D 8

```
public class M {
 public static void
 main(String[] args) {
 int k=0;
 try {
 int i=5/k;
 }
 catch (ArithmeticException e) {
 System.out.print(1);
 }
 catch (RuntimeException e) {
 System.out.print(2);
 return;
 }
 catch (Exception e) {
 System.out.print(3);
 }
 finally {
 System.out.print(4);
 }
 System.out.print(5);
 }
```

```
}
```

Qual è l'output del programma precedente?

- A. 5
- B. 14
- C. 124
- D. 145
- E. 1245
- F. 35

LP1 – Lezione 15

33 / 38

## D 9

```
public class Eccezioni {
 public static void main(String[] args) {
 try {
 if (args.length == 0) return;
 System.out.println(args[0]);
 }
 finally {
 System.out.println("Fine");
 }
 }
}
```

Quali affermazioni riguardanti il precedente programma sono vere?

- A. Se eseguito senza argomenti, il programma non produce output.
- B. Se eseguito senza argomenti, il programma stampa Fine.
- C. Il programma lancia un `ArrayIndexOutOfBoundsException`.
- D. Se eseguito con un argomento, il programma stampa solo l'argomento dato.
- E. Se eseguito con un argomento, il programma stampa l'argomento dato seguito da Fine.

LP1 – Lezione 15

34 / 38

## D 10

Qual è l'output del seguente programma?

```
public class MyClass {
 public static void main(String[] args) {
 RuntimeException re = null;
 throw re;
 }
}
```

- A. Il codice non viene compilato, poiché il `main` non dichiara che lancia una `RuntimeException`.
- B. Il codice non viene compilato, poiché non può rilanciare `re`.
- C. Il programma viene compilato e lancia `java.lang.RuntimeException` in esecuzione.
- D. Il programma viene compilato e lancia `java.lang.NullPointerException` in esecuzione.
- E. Il programma viene compilato, eseguito e termina senza produrre alcun output.

LP1 – Lezione 15

35 / 38

## D 11

Quali di queste affermazioni sono vere?

- A. Se una eccezione non è catturata in un metodo, il metodo termina e viene ripresa la successiva normale esecuzione.
- B. Un metodo sovrapposto in una sottoclasse deve dichiarare che lancia lo stesso tipo di eccezione del metodo che sovrappone.
- C. Il `main` può dichiarare che lancia eccezioni checked.
- D. Un metodo che dichiara di lanciare una certo tipo di eccezione, può lanciare una istanza di una qualunque sottoclasse di quel tipo.
- E. Il blocco `finally` è eseguito se e solo se viene lanciata una eccezione all'interno del corrispondente blocco `try`.

LP1 – Lezione 15

36 / 38

## D 12

```
public class MiaClasse {
 public static void main
 (String[] args) {
 try {
 f();
 }
 catch (MiaEcc e) {
 System.out.print(1);
 throw new RuntimeException();
 }
 catch (RuntimeException e) {
 System.out.print(2);
 return;
 }
 catch (Exception e) {
 System.out.print(3);
 }
 finally {
 System.out.print(4);
 }
 System.out.print(5);
 }
```

```
// MiaEcc e'
// sottoclasse di Exception
static void f() throws MiaEcc {
 throw new MiaEcc();
}
```

Qual è l'output del programma precedente?

- A. 5
- B. 14
- C. 124
- D. 145
- E. 1245
- F. 35

LP1 – Lezione 15

37 / 38

## D 13

```
public class MiaClasse {
 public static void main
 (String[] args)
 throws MiaEcc {
 try {
 f();
 System.out.print(1);
 }
 finally {
 System.out.print(2);
 }
 System.out.print(3);
 }

 // MiaEcc e'
 // sottoclasse di Exception
 static void f() throws MiaEcc {
 throw new MiaEcc();
 }
```

```
}
```

Qual è l'output del programma precedente?

- A. 2 e lancia MiaEcc.
- B. 12
- C. 123
- D. 23
- E. 32
- F. 13

LP1 – Lezione 15

38 / 38

# Linguaggi di Programmazione I – Lezione 11

Prof. Marcello Sette  
mailto://marcello.sette@gmail.com  
http://sette.dnsalias.org

13 maggio 2008

|                                   |           |
|-----------------------------------|-----------|
| <b>Ereditarietà</b>               | <b>3</b>  |
| Specializzazione (1) . . . . .    | 4         |
| Specializzazione (2) . . . . .    | 5         |
| Specializzazione (3) . . . . .    | 6         |
| Polimorfismo . . . . .            | 7         |
| Collezioni eterogenee . . . . .   | 8         |
| Argomenti polimorf. . . . .       | 9         |
| Oper. instanceof . . . . .        | 10        |
| Casting (1) . . . . .             | 11        |
| Casting (2) . . . . .             | 12        |
| Casting (3) . . . . .             | 13        |
| <b>Altre relazioni tra classi</b> | <b>14</b> |
| Composizioni . . . . .            | 15        |
| Aggregazioni . . . . .            | 16        |
| Associazioni . . . . .            | 17        |
| Molteplicità . . . . .            | 18        |
| <b>Overloading e overriding</b>   | <b>19</b> |
| Overl. di metodi . . . . .        | 20        |
| Overl. di costruttori . . . . .   | 21        |
| Overr. di metodi (1) . . . . .    | 22        |
| Overr. di metodi (2) . . . . .    | 23        |
| Overr. di metodi (3) . . . . .    | 24        |
| <b>Costruzione di oggetti</b>     | <b>25</b> |
| super . . . . .                   | 26        |
| super-costruttore . . . . .       | 27        |
| Costr. di oggetti . . . . .       | 28        |
| Esempio (1.a) . . . . .           | 29        |
| Esempio (1.b) . . . . .           | 30        |
| Esempio (2.a) . . . . .           | 31        |
| Esempio (2.b) . . . . .           | 32        |
| <b>La classe Object</b>           | <b>33</b> |
| La classe Object . . . . .        | 34        |

|                              |           |
|------------------------------|-----------|
| Il metodo equals . . . . .   | 35        |
| - Esempio (1) . . . . .      | 36        |
| - Esempio (2) . . . . .      | 37        |
| Il metodo toString . . . . . | 38        |
| <b>Le classi “wrapper”</b>   | <b>39</b> |
| Definizione . . . . .        | 40        |
| Uso . . . . .                | 41        |
| <b>Esercizi</b>              | <b>42</b> |
| Esercizi . . . . .           | 42        |

## Panoramica della lezione

Ereditarietà

Altre relazioni tra classi

Overloading e overriding

Costruzione di oggetti

La classe Object

Le classi "wrapper"

Esercizi

LP1 – Lezione 11

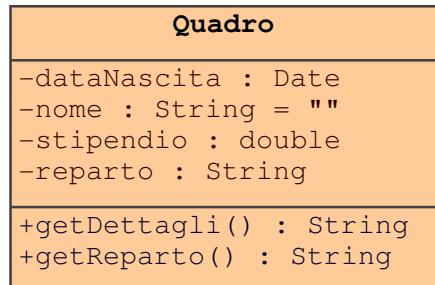
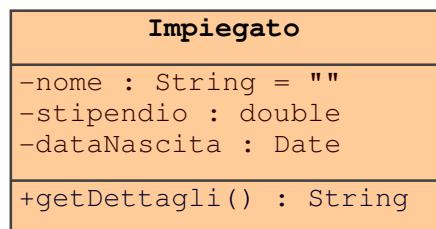
2 / 42

## Ereditarietà

3 / 42

### Specializzazione (1)

Le classi Impiegato e Quadro:



```
public class Impiegato {
 private String nome = "";
 private double stipendio;
 private Date dataNascita;

 public String getDettagli() {
 ...
 }
}
```

```
public class Quadro {
 private String nome = "";
 private double stipendio;
 private Date dataNascita;
 private String reparto;

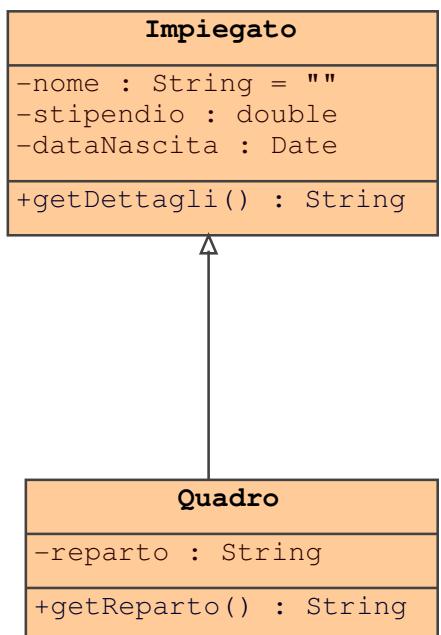
 public String getDettagli() {...}
 public String getReparto() {
 return reparto;
 }
}
```

LP1 – Lezione 11

4 / 42

## Specializzazione (2)

Ridefinizione:



```
public class Impiegato {
 private String nome = "";
 private double stipendio;
 private Date dataNascita;

 public String getDettagli() {
 ...
 }
}

public class Quadro
 extends Impiegato {
 private String reparto;
 public String getReparto() {
 return reparto;
 }
}
```

LP1 – Lezione 11

5 / 42

## Specializzazione (3)

- Una sottoclasse eredita tutti i metodi e le variabili della superclasse.
- Una sottoclasse NON eredita i costruttori della superclasse.
- I due modi, esclusivi uno con l'altro, per includere i costruttori in una classe sono:
  - ◆ usare il costruttore di default (senza parametri);
  - ◆ scrivere uno o più costruttori esplicativi.

LP1 – Lezione 11

6 / 42

## Polimorfismo

- Un oggetto ha solo una forma.
- Polimorfismo 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).
- Poiché si può dire sia:

```
Impiegato impiegato = new Impiegato();
```

sia:

```
Impiegato impiegato = new Quadro();
```

la stessa variabile `impiegato` può essere usata per riferirsi ai due oggetti distinti; l'utilizzatore della variabile in entrambi i casi crederà di riferirsi ad una istanza di `Impiegato`.

- Non sarà, pertanto, mai legale scrivere:

```
String reparto = impiegato.getReparto();
```

anche se `impiegato` fosse in realtà un riferimento ad una istanza di `Quadro`.

LP1 – Lezione 11

7 / 42

## Collezioni eterogenee

- Collezioni *omogenee* di oggetti della stessa classe:

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

- Collezioni *eterogenee* di oggetti di classi diverse:

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

LP1 – Lezione 11

8 / 42

## Argomenti polimorfici

- Si possono costruire metodi che accettino come parametro un riferimento “generico” ed operare in modo automatico su un più vasto insieme di oggetti.
- In questo caso è il metodo ad ignorare la reale natura (forma) dell'oggetto che gli viene specificato come parametro attuale.
- Per esempio, poiché un `Quadro` è un `Impiegato`:

```
// Nella classe A
public double getIrpef(Impiegato i) {
 ...
}

// In un'altra classe:
A a = new A();
Quadro q = new Quadro();

double irpef = a.getIrpef(q)
```

LP1 – Lezione 11

9 / 42

## L'operatore instanceof

- Poiché è lecito scambiarsi oggetti usando riferimenti ai loro antenati (nella gerarchia ad albero), potrebbe essere a volte necessario sapere esattamente la forma dell'oggetto con cui si ha a che fare.
- Per esempio, supponiamo che vi sia una gerarchia di classi:

```
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`. Per saperlo:

```
public void faQualcosa(Impiegato i) {
 if (i instanceof Quadro) {
 // elabora un Quadro
 } else if (i instanceof Segretario) {
 // elabora un Segretario
 } else {
 // elabora un Impiegato qualunque
 }
}
```

## Casting (1)

- Nell'eventualità che 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 rendere visibili tutte le caratteristiche dell'oggetto occorre fare un "cast" esplicito:

```
public void faQualcosa(Impiegato i) {
 if (i instanceof Quadro) {
 Quadro q = (Quadro) i;
 System.out.println(
 "Questo e' il coordinatore del reparto " +
 q.getReparto());
 }
 ...
}
```

- Se non ci fosse il cast, il metodo `getReparto` sarebbe stato inaccessibile al compilatore.

## Casting (2)

Ogni tentativo di effettuare un cast di riferimenti ad oggetti è soggetto a regole precise. A tale scopo, siano A, B e C tre tipi:

- Gli "upcast" sono sempre permessi e, infatti, non richiedono l'operatore di cast. Essi sono realizzati con una semplice assegnazione. Il codice:

```
C c;
...
B b = c;
```

è corretto a patto che C sia sottoclasse di B, indipendentemente dall'oggetto a cui si riferisce c.

## Casting (3)

- Per i “downcast” il compilatore controlla che l’operazione sia almeno possibile: la classe del riferimento di destinazione deve essere una sottoclasse del riferimento di origine. Il codice:

```
B b;
...
C c = (C) b;
```

è corretto in compilazione se C è sottoclasse di B.

- Una volta che il compilatore abbia ammesso l’operazione, essa sarà infine controllata a runtime, per verificare che il tipo dell’oggetto riferito sia compatibile con il tipo del riferimento di destinazione. Se, per esempio, si omettesse il controllo con `instanceof` e si realizzasse un downcast con un oggetto che non è realmente del tipo giusto (quello di destinazione, per intenderci), verrà lanciata una eccezione in esecuzione.
- Le regole complete che governano i cast (poiché coinvolgono anche classi astratte ed interfacce) saranno discusse in una lezione successiva.

## Altre relazioni tra classi

### Composizioni

- La composizione 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.
- Esempio:



```
public class Automobile {
 private Motore motore;
 // altri attributi

 public Automobile (int cilindrata, int numeroCilindri,
 String marca, String modello) {
 motore = new Motore (cilindrata, numeroCilindri);
 // altro codice
 }

 // altri metodi
}
```

## 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 esso. Esempio:



```
public class Automobile {
 private Motore motore;
 // altri attributi

 public Automobile(Motore motore, String marca, String modello) {
 this.motore = motore;
 // altro codice
 }

 // altri metodi
}

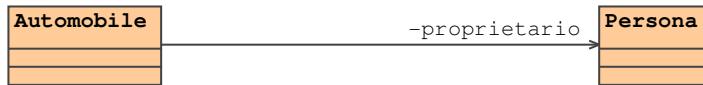
// altrove nel codice
Motore mioMotore = new Motore(1400, 3);
Automobile miaAuto = new Automobile(mioMotore, "Seat", "Ibiza");
```

LP1 – Lezione 11

16 / 42

## Associazioni

- Nella semplice associazione le vite degli oggetti, sia pure correlate, esistono in modo totalmente indipendente.
- Esempio



```
public class Automobile {
 private Persona proprietario;
 // altri attributi

 public void setProprietario(Persona proprietario) {
 this.proprietario = proprietario;
 }

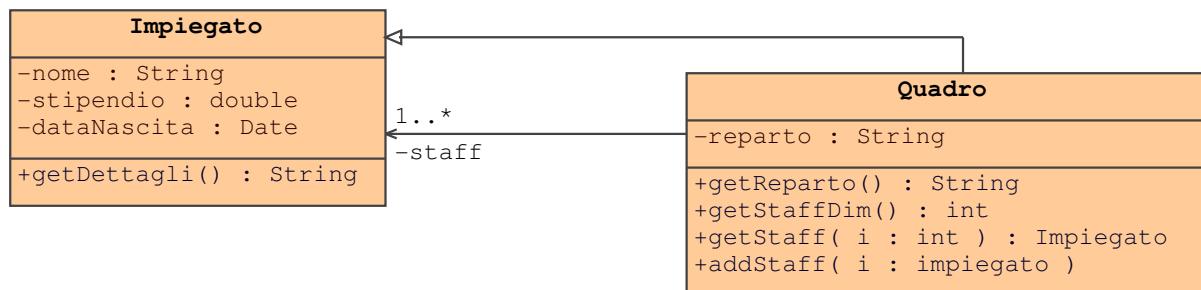
 // altri metodi
}

// altrove nel codice
Automobile miaAuto = new Automobile("Seat", "Ibiza");
Persona io = new Persona("Marcello", "Sette");
miaAuto.setProprietario(io);
```

LP1 – Lezione 11

17 / 42

## Molteplicità



```
public class Quadro extends Impiegato {
 private String reparto = "";
 private Impiegato[] staff = new Impiegato[20];
 private int staffDim = 0;

 public void getReparto() { return reparto }
 public int getStaffDim() { return staffDim }
 public Impiegato getStaff(i:int) { return staff[i] }
 public void addStaff(i:Impiegato) { staff[staffDim++] = i }
}
```

LP1 – Lezione 11

18 / 42

## Overloading e overriding

19 / 42

### Overloading di metodi

- Metodi **con lo stesso nome, nella stessa classe**, che svolgono lo stesso compito con diversi argomenti.
- Per esempio:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

- La lista degli argomenti **DEVE** essere diversa, perché essa sola è usata per distinguere i metodi.
- Il tipo di ritorno **PUÒ** essere diverso ma non è usato per distinguere i metodi (cioè, metodi con uguale nome e lista di parametri, ma diverso tipo di ritorno sono indistinguibili ed il compilatore segnala l'errore).

LP1 – Lezione 11

20 / 42

## Overloading di costruttori

- Stesse regole dei metodi.
- Il riferimento `this` può essere usato, NELLA PRIMA LINEA di un costruttore, per invocare un altro costruttore.

```
public class Impiegato {
 private static final double STIPENDIO_BASE = 15000.00;
 private String nome; private double stipendio;
 private Date dataNascita;

 public Impiegato(String nome, double stip, Date nasc) {
 this.nome=nome; stipendio=stip; dataNascita=nasc;
 }
 public Impiegato(String nome, double stip) {
 this(nome, stip, null);
 }
 public Impiegato(String nome, Date nasc) {
 this(nome, STIPENDIO_BASE, nasc);
 }
 public Impiegato(String nome) {
 this(nome, STIPENDIO_BASE);
 }
 // Altro codice
}
```

## Overriding di metodi (1)

Una sottoclasse può sovrapporre un metodo ereditato e visibile, a patto che, rispetto al metodo della superclasse, il nuovo metodo abbia (a) lo stesso nome, tipo di ritorno e lista di argomenti; (b) visibilità non inferiore [vedremo poi].

```
public class Impiegato {
 protected String nome; protected double stipendio;
 protected Date dataNascita;

 public String getDettagli() {
 return "Nome: " + nome + "\n" +
 "Stipendio: " + stipendio;
 }

 public class Quadro extends Impiegato {
 protected String reparto;

 public String getDettagli() {
 return "Nome: " + nome + "\n" +
 "Stipendio: " + stipendio + "\n" +
 "Reparto: " + reparto;
 }
 }
}
```

## Overriding di metodi (2)

- In riferimento all'esempio precedente, è chiaro quali metodi sono invocati in:

```
Impiegato i = new Impiegato();
Quadro q = new Quadro();

System.out.println(i.getDettagli());
System.out.println(q.getDettagli());
```

- Ma quale metodo è invocato qui?

```
Impiegato i = new Quadro();

System.out.println(i.getDettagli());
```

- Si ottiene il comportamento del tipo che assumerà la variabile in esecuzione e non il comportamento del tipo che la variabile ha in compilazione (invocazione virtuale di metodo).
- L'invocazione virtuale dei metodi è la massima rivelazione del polimorfismo.

## Overriding di metodi (3)

Riguardo al vincolo sulla non inferiore visibilità, il codice seguente non viene compilato (a dispetto del fatto che la vera risoluzione di tipo avviene in esecuzione):

```
public class Padre {
 public void faQualcosa() { }
}

public class Figlio extends Padre {
 private void faQualcosa() { }
}

public class UsaEntrambi {
 public void faAltro() {
 Padre p1 = new Padre();
 Padre p2 = new Figlio();
 p1.faQualcosa();
 p2.faQualcosa();
 }
}
```

## La parola chiave super

- `super` è usata in una classe per riferirsi alla superclasse.
- `super` è usata per riferirsi ai membri della superclasse (attributi e metodi) con la dot notation.
- I membri della superclasse accessibili con `super` sono anche quelli che la superclasse ha ereditato e non solo quelli esplicitamente definiti in essa.
- NON si può usare `super.super.membro` per accedere a membri di classi di livello superiore alla prima superclasse.

```
public class Impiegato {
 private String nome;
```

```
private double stipendio;

public String getDettagli() {
 return "Nome: " + nome +
 "\nStipendio: " + stipendio;
}
}

public class Quadro
 extends Impiegato {
 private String reparto;

 public String getDettagli() {
 return
 super.getDettagli() +
 "\nReparto: " + reparto;
 }
}
```

LP1 – Lezione 11

26 / 42

## 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 (e.g. `this(...)`).
- Se, come prima linea nel proprio costruttore, non c'è né `this`, né `super`, il compilatore pone una chiamata隐式的 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.

```
public class Quadro
 extends Impiegato {
 private String reparto;

 public Quadro(String nome,
 double stipendio,
 String reparto) {
 super(nome, stipendio);
 this.reparto = reparto;
 }
 public Quadro(String nome,
 String reparto) {
 super(nome);
 this.reparto = reparto;
 }
 public Quadro(String reparto) {
 this.reparto = reparto;
 // errore: assenza del
 // costruttore Impiegato()
 // nella superclasse
 }
}
```

LP1 – Lezione 11

27 / 42

## Costruzione ed inizializzazione di oggetti

Riprendiamo

- 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`, poiché `Object` non ha superclasse.
  4. Esegue ogni inizializzazione esplicita delle variabili di istanza.
  5. Esegue il corpo del costruttore corrente.

LP1 – Lezione 11

28 / 42

### Esempio (1.a)

Esecuzione di `new Quadro("M 7", "vendite")`.

```
public class Impiegato {
 private String nome;
 private double stipendio=15000.00;
 private Date dataNascita;

 public Impiegato(String n, Date d) {
 nome = n;
 dataNascita = d;
 }
 public Impiegato(String n) {
 this(n, null);
 }
}

public class Quadro extends Impiegato {
 private String reparto;

 public Quadro(String n, String r) {
 super(n); // e se manca sta riga?
 reparto = r;
 }
}
```

LP1 – Lezione 11

29 / 42

## Esempio (1.b)

- 0 Inizializzazione base (`new ...`).
  - 0.1 Allocazione memoria per un oggetto Quadro.
  - 0.2 Inizializzazione di reparto al valore di default `null`.
- 1 Esecuzione del costruttore: `Quadro("M 7", "vendite")`.
  - 1.1 Inizializzazione dei parametri del costruttore: `n="M 7", r="vendite"`.
  - 1.2 Non c'è chiamata a `this(...)`.
  - 1.3 Invocazione di `super("M 7")` come istanza di `Impiegato(String n)`.
    - 1.3.1 Inizializzazione del parametro del costruttore: `n="M 7"`.
    - 1.3.2 Esecuzione di `this("M 7", null)` come istanza di `Impiegato(String n, Date d)`.
      - 1.3.2.1 Inizializzazione dei parametri del costruttore: `n="M 7", d=null`.
      - 1.3.2.2 Nessuna chiamata esplicita a `this(...)`.
      - 1.3.2.3 Nessuna chiamata esplicita a `super(...)` (`Object` è la radice).
      - 1.3.2.4 Nessuna inizializzazione esplicita.
      - 1.3.2.5 Nessun corpo da eseguire.
    - 1.3.2.4 Inizializzazione esplicita della variabile: `stipendio=15000.00`.
    - 1.3.2.5 Esecuzione del corpo del costruttore: `nome="M 7", dataNascita=null`.
  - 1.3.3 Saltato.
  - 1.3.4 Saltato.
  - 1.3.5 Nessun corpo in `Impiegato(String n)`.
- 1.4 Nessuna inizializzazione esplicita per Quadro.
- 1.5 Esecuzione di `reparto="Vendite"`.

LP1 – Lezione 11

30 / 42

## Esempio (2.a)

```
public class Impiegato {
 private String nome;
 private double
 stipendio=15000.00;
 private Date dataNascita;
 private String sommario;

 public Impiegato(String n,
 Date d) {
 nome = n; dataNascita = d;
 sommario = getDettagli();
 }
 public Impiegato(String n) {
 this(n, null);
 }

 public String getDettagli(){
 return "Nome: " + nome +
 "\nStipendio: " + stipendio
 + "\nData di nascita: " +
 dataNascita;
 }
}

public class Quadro
 extends Impiegato {
 private String reparto;

 public Quadro(String n,
 String r) {
 super(n);
 reparto = r;
 }

 public String getDettagli(){
 return super.getDettagli() +
 "\nReparto: " + reparto;
 }
}
```

LP1 – Lezione 11

31 / 42

## Esempio (2.b)

Quello che succede, nella generazione di una istanza di Quadro mediante new Quadro("M 7", "Vendite"), è che:

- il metodo getDettagli() viene sovrapposto nella classe Quadro;
- viene invocato getDettagli() nel costruttore della superclasse Impiegato;
- a runtime viene scelto il metodo di Quadro (invocazione virtuale di metodi);
- la variabile reparto non è stata, però, ancora esplicitamente inizializzata: piccolo problema, perché il programma viene comunque eseguito, ma viene aggiunta alla stringa di sommario "Reparto: null";
- tutto ciò avrebbe potuto essere catastrofico, nel caso in cui la variabile fosse stata non una variabile di heap, ma una variabile di stack (in questo caso neppure implicitamente inizializzata).

Regola aurea: **Se si invoca un metodo in un costruttore, rendere quel metodo privato!**

In questo modo il metodo non può risultare sovrapposto nella sottoclasse, perché essa pure ereditandolo, non può vederlo!!!

LP1 – Lezione 11

32 / 42

## La classe Object

33 / 42

### La classe Object

- In Java la classe Object è la radice di tutte le classi.
- Una dichiarazione di classe senza la clausola extends, implicitamente usa "extends Object".  
Pertanto

```
public class Impiegato {
 ...
}
```

è equivalente a

```
public class Impiegato extends Object {
 ...
}
```

- Questo ci permette di sovrapporre numerosi metodi della classe Object, due dei quali saranno discussi qui di seguito.

LP1 – Lezione 11

34 / 42

### Il metodo equals

- L'operatore == determina se due riferimenti sono identici (cioè se 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 ==. 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. Per esempio la classe String realizza il proprio equals, confrontando le stringhe carattere per carattere.
- Viene raccomandato di sovrapporre, insieme con equals, anche il metodo hashCode. Una decente, anche se povera, realizzazione potrebbe usare l'XOR bit a bit dei codici hash degli oggetti in esame.

LP1 – Lezione 11

35 / 42

## - Esempio (1)

```
public class Impiegato {
 private String nome;
 private MiaData nascita;
 private double stipendio;

 public Impiegato(String n, Date d, double s) {
 nome=n; dataNascita=d; stipendio=s;
 }

 public boolean equals(Object o) {
 boolean risultato = false;
 if ((o != null) && (o instanceof Impiegato)) {
 Impiegato i = (Impiegato) o;
 if (nome.equals(i.nome) && nascita.equals(i.nascita)) {
 risultato = true;
 }
 }
 return risultato;
 }

 public int hashCode() {
 return (nome.hashCode() ^ nascita.hashCode());
 }
}
```

## - Esempio (2)

```
public class TestEquals {
 public static void main(String[] args) {
 Impiegato i1 = new Impiegato("Eliana",
 new MiaData(23, 4, 1964),
 25000,00);
 Impiegato i2 = new Impiegato("Marcello",
 new MiaData(23, 4, 1964),
 25000,00);

 System.out.println("i1 identico ad i2: " +
 (i1 == i2));
 System.out.println("i1 equivalente ad i2: " +
 i1.equals(i2));

 i2 = i1;
 System.out.println("\nPoniamo i2=i1");
 System.out.println("i1 identico ad i2: " +
 (i1 == i2));
 }
}
```

## Il metodo `toString`

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

```
Date now = new Date();
System.out.println(now);
```

è rozzamente equivalente a:

```
Date now = new Date();
System.out.println(now.toString());
```

- Occorre sovrapporre 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".

## Le classi "wrapper"

### Definizione

- I tipi primitivi non sono oggetti. Se si vogliono manipolare come oggetti, è possibile avvolgere un SINGOLO valore con una opportuno oggetto, cosiddetto "wrapper". La corrispondenza è questa:

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

## Uso

- Si può costruire un oggetto wrapper, passando il relativo valore al costruttore appropriato. Per esempio:

```
int i = 500;
Integer indice = new Integer(i);
int x = indice.intValue();
```

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

```
int x = Integer.valueOf(str).intValue();
int y = Integer.parseInt(str);
```

## Esercizi

### Esercizi

1. In questo esercizio verranno create due sottoclassi della classe Conto nel progetto banca: ContoCorrente e LibrettoRisparmio.  
Si dovrà sovrapporre il metodo preleva in ContoCorrente ed usare super per invocare il costruttore della superclasse.
2. Come ulteriore continuazione del progetto di gestione bancaria, per ciascun cliente della banca in questo esercizio verrà creata una collezione eterogenea di conti (max. 5), cioè lo stesso cliente può essere titolare di conti di diverso tipo.

# Linguaggi di Programmazione I – Lezione 8

Prof. Marcello Sette  
mailto://marcello.sette@gmail.com  
http://sette.dnsalias.org

29 aprile 2008

|                                    |           |
|------------------------------------|-----------|
| <b>Identifieri e parole chiavi</b> | <b>3</b>  |
| Commenti . . . . .                 | 4         |
| Blocchi . . . . .                  | 5         |
| Identifieri . . . . .              | 6         |
| Parole chiavi. . . . .             | 7         |
| Esempi . . . . .                   | 8         |
| <b>Tipi primitivi</b>              | <b>9</b>  |
| Elenco . . . . .                   | 10        |
| boolean . . . . .                  | 11        |
| char . . . . .                     | 12        |
| E le stringhe? . . . . .           | 13        |
| Tipi interi . . . . .              | 14        |
| Litterali interi . . . . .         | 15        |
| Tipi a virgola mobile . . . . .    | 16        |
| Litterali a . . . . .              | 17        |
| Esempio. . . . .                   | 18        |
| <b>Tipi reference</b>              | <b>19</b> |
| Cosa sono . . . . .                | 20        |
| Costruzione di . . . . .           | 21        |
| - Allocazione di . . . . .         | 22        |
| - Inizializzazione . . . . .       | 23        |
| - Esecuzione del . . . . .         | 24        |
| - Assegnazione . . . . .           | 25        |
| E non è tutto! . . . . .           | 26        |
| Esempio. . . . .                   | 27        |
| <b>Parametri</b>                   | <b>28</b> |
| Passaggio per valore . . . . .     | 29        |
| Esempio. . . . .                   | 30        |
| <b>Il riferimento this</b>         | <b>31</b> |
| Il riferimento this . . . . .      | 31        |
| Esempio (1) . . . . .              | 32        |
| Esempio (2) . . . . .              | 33        |

|                               |           |
|-------------------------------|-----------|
| <b>Convenzioni sul codice</b> | <b>34</b> |
| Convenzioni varie . . . . .   | 34        |
| <b>Esercizi</b>               | <b>35</b> |
| Esercizi . . . . .            | 35        |

## Panoramica della lezione

Identifieri e parole chiavi

Tipi primitivi

Tipi reference

Parametri

Il riferimento this

Convenzioni sul codice

Esercizi

LP1 – Lezione 8

2 / 35

## Identifieri e parole chiavi

3 / 35

### Commenti

Tre stili di scrittura di commento al codice:

```
// commento su una singola riga
```

```
/* commento su una
o piu' righe */
```

```
/** commento usato per la documentazione automatica
del codice */
```

Il formato di quest'ultimo commento e l'uso dello strumento javadoc è discusso nella cartella guide/javadoc della documentazione delle API per Java 2 SDK.

LP1 – Lezione 8

4 / 35

### Blocchi

- Un *enunciato* è costituito da una o più righe di codice terminate da un ' ';' :

```
totale = a + b + c
+ d + e + f;
```

- Un *blocco* è la collezione di enunciati racchiusi tra parentesi graffe:

```
{
 x = y + 1;
 y = x + 1;
}
```

- I blocchi possono essere annidati uno nell'altro.
- Il numero di spazi o righe bianche sono ininfluenti.

LP1 – Lezione 8

5 / 35

## Identifieri

- Sono nomi assegnati a variabili, classi, metodi.
- Possono cominciare con un carattere Unicode, underscore(\_), oppure il dollaro (\$).
- Sono *case sensitive* e non hanno lunghezza massima.
- Esempi:

```
identificatore
userName
user_name
_sys_var1
$change
```

- Il nome di una classe è costituito solo da caratteri ASCII poiché molti sistemi non supportano Unicode.

## Parole chiavi

|          |          |            |           |              |           |
|----------|----------|------------|-----------|--------------|-----------|
| abstract | continue | float      | long      | short        | transient |
| boolean  | default  | for        | native    | static       | true      |
| break    | do       | goto       | new       | strictfp     | try       |
| byte     | double   | if         | null      | super        | void      |
| case     | else     | implements | package   | switch       | volatile  |
| catch    | extends  | import     | private   | synchronized | while     |
| char     | false    | instanceof | protected | this         |           |
| class    | final    | int        | public    | throw        |           |
| const    | finally  | interface  | return    | throws       |           |

- Non possono essere usate come identifieri.
- true, false, null sono in minuscolo, non in maiuscolo come in C++. Strettamente parlando sono litterali, non parole chiavi.
- Non c'è l'operatore sizeof: la dimensione e

la rappresentazione dei tipi è fissa e non dipende dalla realizzazione della JVM.

- goto e const sono parole chiavi che non sono usate in Java.

## Esempi

```
foobar // legale
BIGinterface // legale; contiene parola chiave
$guadagniMenoSpese // legale
3_node5 // illegale: comincia con cifra
!ilCubo // illegale: deve cominciare con
 // lettera, $, o _
```

## Elenco

Otto tipi primitivi:

- Logici: boolean
- Testuali: char
- Interi: byte, short, int, long
- Floating point: float, double

LP1 – Lezione 8

10 / 35

## boolean

- Il tipo boolean ha due littorali: true, false.
- Esempio:

```
boolean ok = true;
```
- Non c'è cast tra tipi interi e boolean. Interpretare valori numerici come valori logici non è permesso in Java.

LP1 – Lezione 8

11 / 35

## char

- Rappresenta un carattere Unicode (16 bit).
- I littorali di questo tipo sono inclusi tra apici singoli (' ').
- Esempi:

```
'a' // la lettera a
'\t' // una tabulazione
'\u03A6' // la lettera greca phi
```

- Le stringhe non sono tipi primitivi.
- Fare riferimento alle specifiche del linguaggio Java per ulteriori codici '\?'.

LP1 – Lezione 8

12 / 35

## E le stringhe?

- String NON È UN TIPO PRIMITIVO, è una classe (comincia per lettera maiuscola).
- "Ha i suoi littorali racchiusi tra apici doppi".
- Può essere usata come segue:

```
String saluto = "Buon giorno!! \n";
String messaggioErrore = "File not found!";
```

- Non è finita qui ...

LP1 – Lezione 8

13 / 35

## Tipi interi

Rappresentano i valori:

| Lunghezza | Tipo  | Range                      |
|-----------|-------|----------------------------|
| 8 bit     | byte  | $-2^7 \dots 2^7 - 1$       |
| 16 bit    | short | $-2^{15} \dots 2^{15} - 1$ |
| 32 bit    | int   | $-2^{31} \dots 2^{31} - 1$ |
| 64 bit    | long  | $-2^{63} \dots 2^{63} - 1$ |

LP1 – Lezione 8

14 / 35

## Litterali interi

- I litterali hanno tre forme: decimale, ottale ed esadecimale.
  - ◆ 2 il valore decimale è due.
  - ◆ 077 lo zero iniziale denota un valore ottale.
  - ◆ 0xBAAC la parte 0x iniziale denota un valore esadecimale.
- Assumono come tipo di default int.
- Suffisso L oppure l se si vuole che siano di tipo long.
  - ◆ 2L è due, rappresentato come long.
  - ◆ 077L è un valore ottale, rappresentato come long.
  - ◆ 0xBAACL è un valore esadecimale, rappresentato come long.
- Quando si assegna il *valore di un litterale* ad una variabile, il compilatore determina la dimensione del litterale a seconda della variabile:

```
short s = 9; // questo e' ok
```

- Quando si assegna il *valore di una espressione* ad una variabile, la dimensione del valore non è modificabile:

```
short s1 = 9 + s; // questo causa errore di compilazione
 // perche' 9 + s e' int non short
```

LP1 – Lezione 8

15 / 35

## Tipi a virgola mobile

Rappresentano i valori:

| Lunghezza | Tipo   |
|-----------|--------|
| 32 bit    | float  |
| 64 bit    | double |

secondo lo standard (IEEE) 754.

LP1 – Lezione 8

16 / 35

## Litterali a virgola mobile

- Hanno come tipo di default il tipo double.
- Suffisso F o f per litterali del tipo float.
- Suffisso D o d per litterali del tipo double.
- Esempi
  - ◆ 3.14 un semplice valore a virgola mobile (double).
  - ◆ 6.02E23 un altro valore.
  - ◆ 2.718F un semplice float.
  - ◆ 123.4E-5D un double con una D superflua.

LP1 – Lezione 8

17 / 35

## Esempio

```
public class Assign {
 public static void main (String args []) {
 int x, y;
 float z = 3.1415f;
 double w = 3.14145;
 boolean truth = true;
 char c;
 String str;
 String str1 = "ciao";
 c = 'A';
 str = "ciao a tutti";
 x = 6;
 y = 1000;
 }
}
```

Queste sono assegnazioni illegali:

```
y = 3.14159; // 3.14159 non e' un int; richiede casting
w = 175,000; // la virgola invece del punto decimale
truth = 1; // errore comune tra programmatore C / C++
z = 3.14159; // 3.14159 non e' un float; richiede casting
```

LP1 – Lezione 8

18 / 35

## Cosa sono

- Tutti i tipi non primitivi sono tipi *reference*.
- Una variabile *reference* contiene la “maniglia” di un oggetto.
- Esempio:

```
public class MiaData {
 private int giorno = 1;
 private int mese = 1;
 private int anno = 2006;
}
```

La classe *MiaData* può essere usata in questo modo:

```
public class TestMiaData {
 public static void main (String[] args) {
 MiaData oggi = new MiaData();
 }
}
```

## Costruzione di oggetti

- `new Xxx()` serve ad allocare spazio per il nuovo oggetto. Scatena i seguenti processi:
  1. Viene allocato lo spazio per il nuovo oggetto e le variabili dell’istanza sono inizializzate al loro valore di default (e.g. 0, `false`, `null`, e così via).
  2. Viene eseguita ogni inizializzazione esplicita degli attributi.
  3. Viene eseguito un costruttore.
  4. Viene assegnato il riferimento finale all’oggetto.
- Esaminiamo separatamente ciascuna di queste fasi, mostrando ciò che succede quando viene eseguito il codice:

```
MiaData nascita = new MiaData (23, 4, 1964);
```

## - Allocazione di memoria

Nell'enunciato:

```
MiaData nascita = new MiaData (23, 4, 1964);
```

- la dichiarazione `MiaData nascita` causa l'allocazione dello spazio memoria del solo riferimento (non ancora inizializzato):

nascita

????

- l'uso della parola chiave `new`, alloca spazio per `MiaData` (inizializzata ai valori di default – vediamo poi):

nascita

????

giorno

0

mese

0

anno

0

## - Inizializzazione degli attributi

- Successivamente, per gli attributi sono usate le inizializzazioni esplicite all'interno della classe, quelle che eventualmente sono scritte nel momento della definizione dell'attributo (inizializzazione dell'oggetto da parte del progettista della classe):

nascita

????

giorno

1

mese

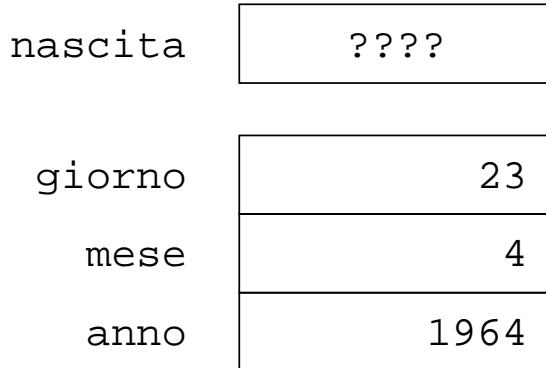
1

anno

2006

### - Esecuzione del costruttore

- Ora è invocato il costruttore. Con esso si possono sostituire inizializzazioni personali dell'utente dell'oggetto a quelle di default previste dal progettista della classe. Si possono anche passare argomenti, così che il codice che richiede le costruzione del nuovo oggetto possa controllare l'oggetto che verrà creato:

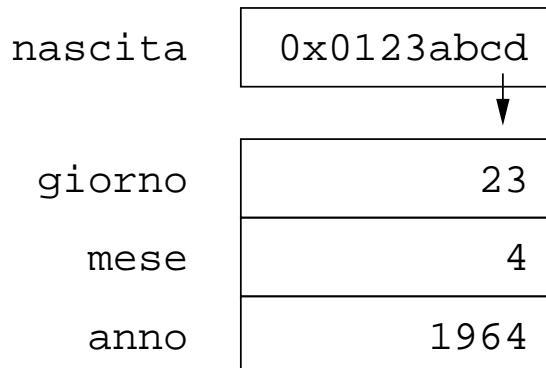


LP1 – Lezione 8

24 / 35

### - Assegnazione del riferimento

- L'assegnazione, infine, inserisce l'indirizzo del nuovo oggetto nella locazione del riferimento:



LP1 – Lezione 8

25 / 35

### E non è tutto!

Purtroppo il processo di costruzione di oggetti e della loro inizializzazione è notevolmente più complesso di come è stato descritto qui.

Rituneremo successivamente su questo argomento.

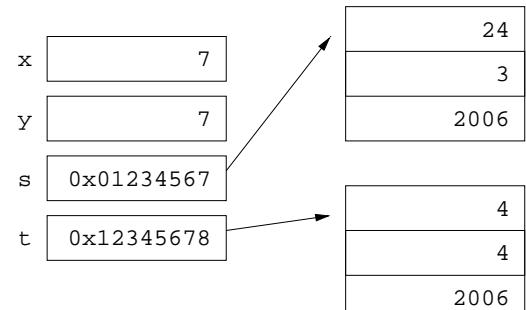
LP1 – Lezione 8

26 / 35

## Esempio

```
int x = 7;
int y = x;

MiaData s = new MiaData(24, 3, 2006);
MiaData t = s;
t = new MiaData(4, 4, 2006);
```



LP1 – Lezione 8

27 / 35

## Parametri

28 / 35

### Passaggio per valore

- Java permette il passaggio dei parametri per valore (nella nostra tassonomia, 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.
- Attenzione che, in quest'ultimo caso, sarà possibile la modifica nel contesto del chiamante (mai del valore della variabile) dell'oggetto a cui fa riferimento quella variabile.

LP1 – Lezione 8

29 / 35

## Esempio

```
public class PassTest {
 public static void
 cambiaValore(int v) {
 v = 55;
 }
 public static void
 cambiaOggetto(MiaData d) {
 d = new MiaData(1, 1, 2005);
 }
 public static void
 cambiaAttributo(MiaData d) {
 d.setGiorno(16);
 }

 public static void
 main(String args[]) {
 MiaData data =
 new MiaData(24, 4, 2006);
 int val = 11;
```

```
cambiaValore(val);
System.out.println(
 "val vale: " + val);

cambiaOggetto(data);
data.print();

cambiaAttributo(data);
data.print();
}
```

L'output è:

```
$ java PassTest
val vale: 11
MiaData: 24-4-2006
MiaData: 16-4-2006
```

LP1 – Lezione 8

30 / 35

### Il riferimento this

La parola chiave this 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'oggetto.
- Per permettere ad un oggetto di passare il riferimento a sè stesso come parametro ad un altro metodo o costruttore.

L'esempio seguente mostra le tecniche precedenti.

LP1 – Lezione 8

31 / 35

### Esempio (1) – La classe MiaData

```
public class MiaData {
 private int giorno = 1;
 private int mese = 1;
 private int anno = 2006;

 public MiaData (int giorno,
 int mese,
 int anno) {
 this.giorno = giorno;
 this.mese = mese;
 this.anno = anno;
 }

 public MiaData(MiaData data) {
 giorno = data.giorno;
 mese = data.mese;
 anno = data.anno;
 }

 public MiaData addGiorni
 (int g) {
 MiaData data =
 new MiaData(this);

 data.giorno = data.giorno + g;
 // Non ben realizzato ...

 return data;
 }

 public void print() {
 System.out.println(
 "MiaData: " + giorno +
 "-" + mese + "-" + anno);
 }
}
```

LP1 – Lezione 8

32 / 35

### Esempio (2) – La classe TestMiaData

Per provare il funzionamento, scriviamo:

```
public class TestMiaData {
 public static void main(String[] args) {
 MiaData nascita = new MiaData(23, 4, 1964);
 MiaData la_settimana_dopo = nascita.addGiorni(7);

 la_settimana_dopo.print();
 }
}
```

Quindi:

```
$ java TestMiaData
MiaData: 30-4-1964
```

LP1 – Lezione 8

33 / 35

## Convenzioni varie

- Pacchetti:

```
package oggetti.geometria;
```

- Classi:

```
class Circonferenza;
```

- Interfacce:

```
interface FiguraPiana;
```

- Metodi:

```
getOffset();
```

- Variabili:

```
dimensioneX
```

- Costanti:

```
PI_GRECO
```

- Parentesi:

```
if (condition) {
 do something
} else {
```

```
 do something else
}
```

- Spaziatura: un solo enunciato per riga; due spazi di indenting per i blocchi annidati.

- Commenti:

```
// Un commento su una riga

/* Commenti su piu' righe
 ehgir 'uip us itnemmoC */

/** Commento per documentazione
 * automatica.
 * @see Altra classe per
 * maggiori informazioni
 */
```

## Esercizi

### Esercizi

1. Sulla creazione e uso degli oggetti.
2. Estensione dell'esercizio della lezione precedente (pacchetto banca).

# Addendum a lezione 3

Implementazione efficiente dell'ambiente non  
locale con scoping statico

Piero Bonatti

# Annidamento (nesting)

```
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 esempio:
  - 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 la contengono
  - il livello di nesting di q e s è 1
  - quello di r è 2
- **L'ambiente statico non locale** di una procedura è dato dall'ambiente delle procedure in cui è innestata

# Annidamento (nesting)

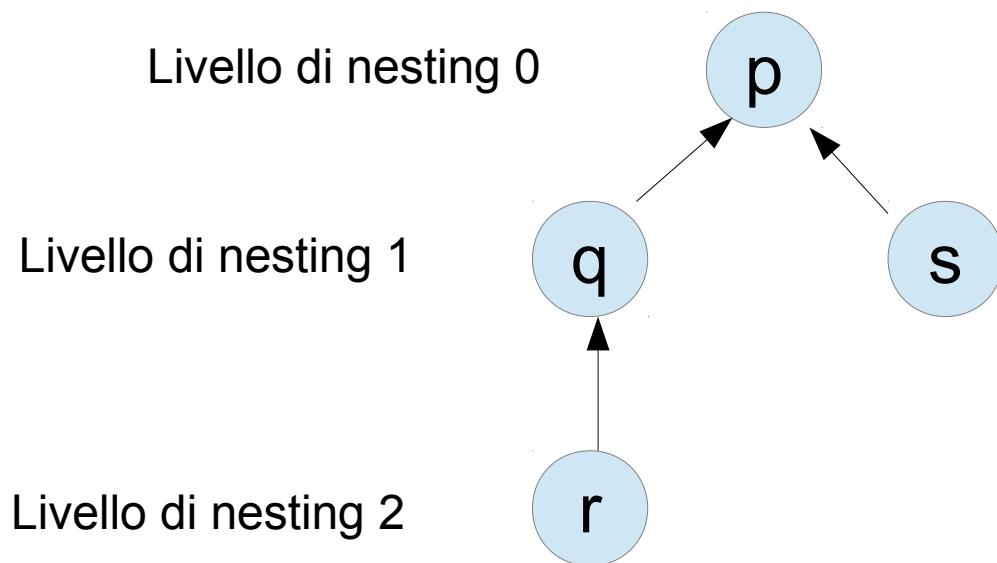
```
program p;
var a,b,c int;

procedure q;
var a,c int;

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

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

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



Le frecce indicano l'ambiente non locale

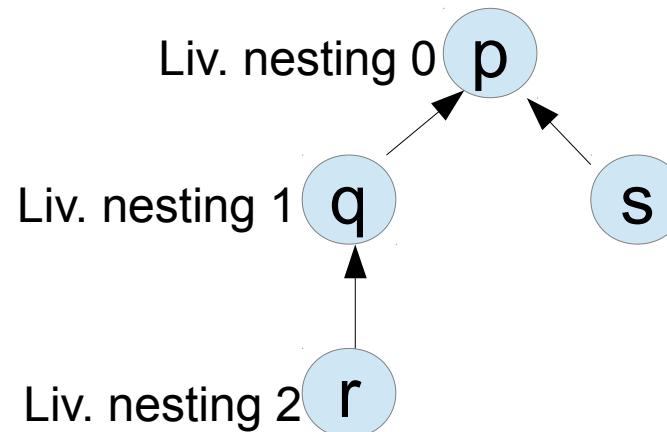
# Rappresentazione variabili non locali

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

procedure q;
var a,c int;

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

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

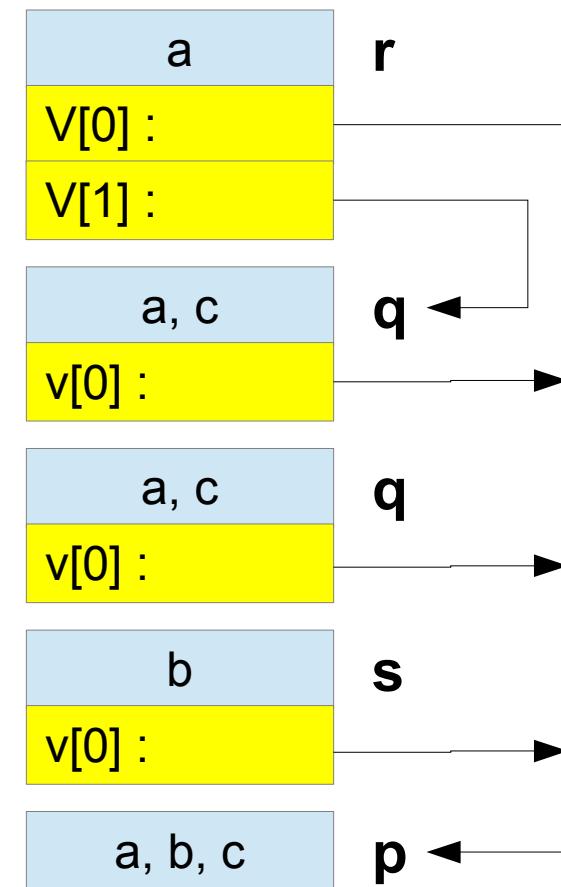


Rappresentazione variabili non locali:  
*(livello, offset)*

Nella procedura r:  
b è rappresentata da (0,2)  
c è rappresentata da (1,2)

$\text{env}(x) = v[\text{livello}] + \text{offset}$

Stack di attivazione



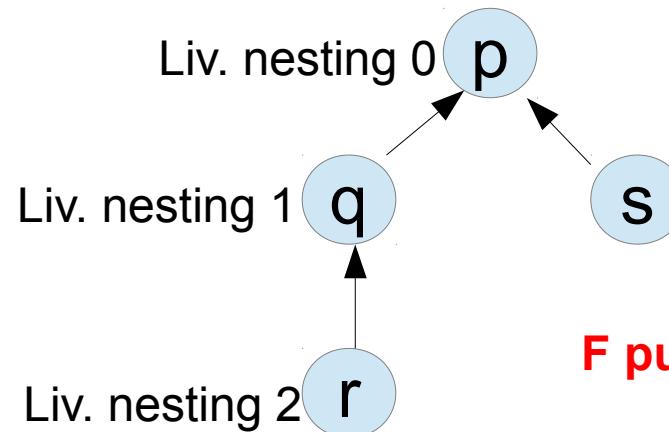
# Visibilità delle procedure

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

procedure q;
var a,c int;

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

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



(Ambito statico)

**F può chiamare G se:**

- G è definita in F
- G è definita in uno dei blocchi che contiene F

**quindi:**

**F e G hanno sempre una parte di ambiente non locale in comune**

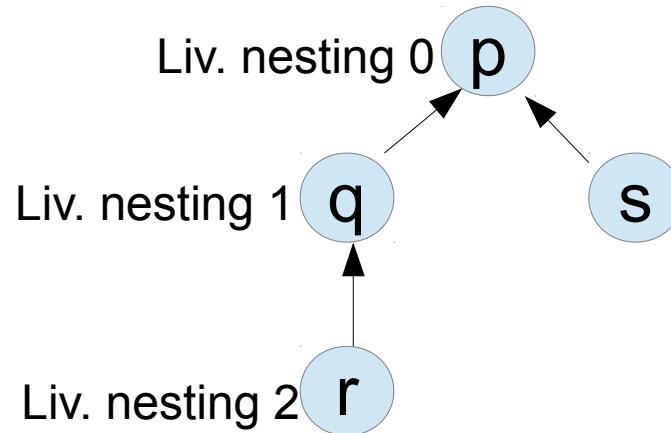
# Mantenimento del vettore degli ambienti non locali

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

procedure q;
var a,c int;

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

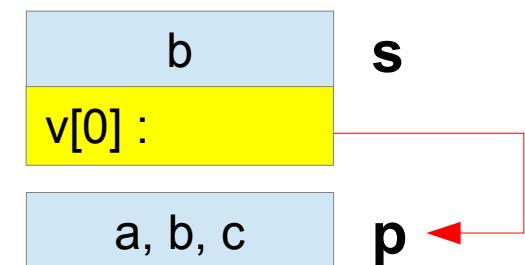
procedure s;
var b int;
...
...
```



Chiamata a procedure definite localmente:

**Esempio 1: p chiama s:**

- Aumenta liv. di nesting
- Aggiungere elem. a v[ ]



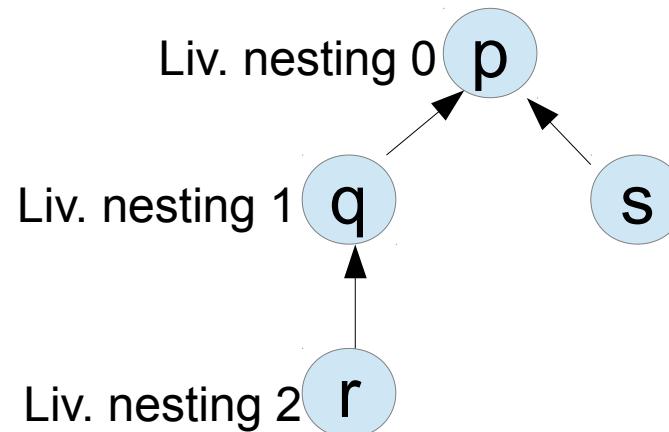
# Mantenimento del vettore degli ambienti non locali

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

procedure q;
var a,c int;

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

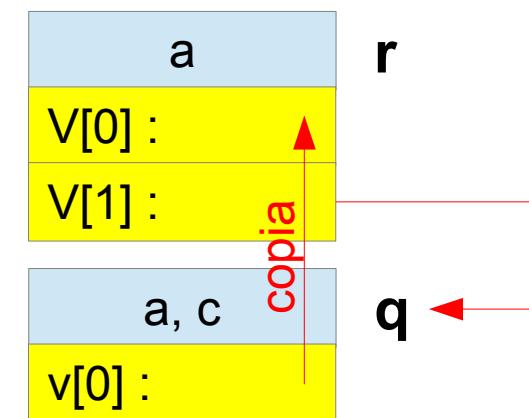
procedure s;
var b int;
...
...
```



Chiamata a procedure definite localmente:

Esempio 2: q chiama r:

- Aumenta liv. di nesting
- Aggiungere elem. a v[ ]
- Copiare gli altri elementi



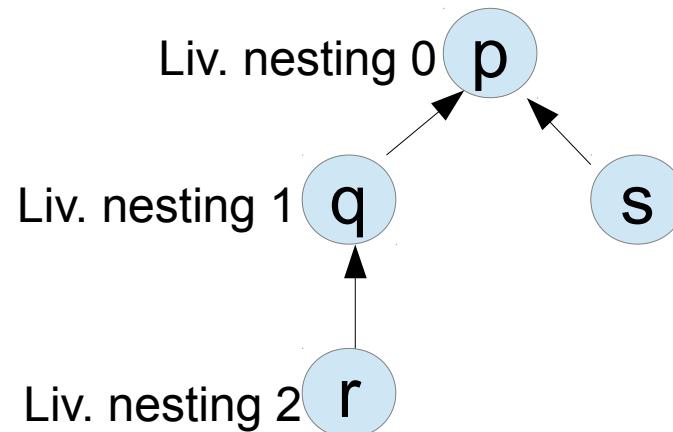
# Mantenimento del vettore degli ambienti non locali

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

procedure q;
var a,c int;

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

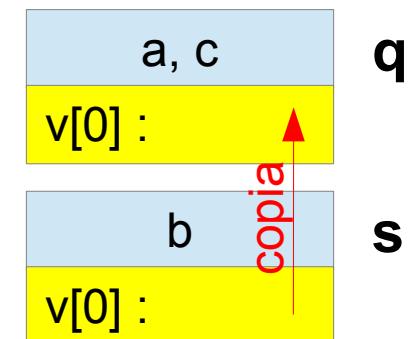
procedure s;
var b int;
...
...
```



Chiamata a procedure definite **esternamente**:

**Esempio 1: s chiama q:**

- stesso liv. di nesting
- Copiare gli elementi di v[ ]



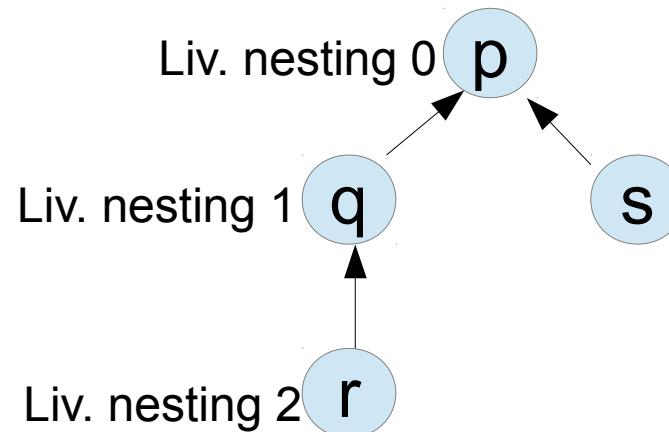
# Mantenimento del vettore degli ambienti non locali

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

procedure q;
var a,c int;

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

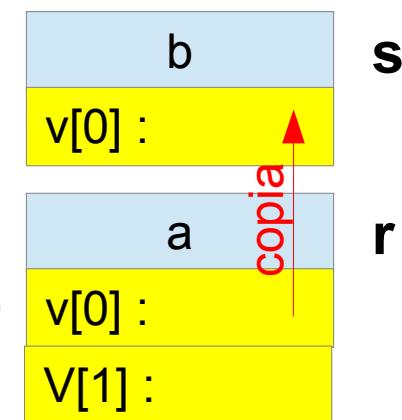
procedure s;
var b int;
...
...
```



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



# Proprietà di questa implementazione

- Accesso alle variabili in tempo costante
  - Accesso a vettore + somma del puntatore ivi contenuto e dell'offset
  - Indipendente dai livelli 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 naïve 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

# Linguaggi di Programmazione I – Lezione 7

Prof. Marcello Sette  
mailto://marcello.sette@gmail.com  
http://sette.dnsalias.org

22 aprile 2008

|                                  |           |
|----------------------------------|-----------|
| <b>Introduzione a Java</b>       | <b>3</b>  |
| Tecnologia Java . . . . .        | 4         |
| <b>Un esempio</b>                | <b>5</b>  |
| I sorgenti . . . . .             | 6         |
| Compilazione ed . . . . .        | 7         |
| Errori comuni . . . . .          | 8         |
| L'esempio esempl. . . . .        | 9         |
| <b>JVM e JRE</b>                 | <b>10</b> |
| JVM . . . . .                    | 11        |
| Punti di vista . . . . .         | 12        |
| Garbage Collector . . . . .      | 13        |
| JRE . . . . .                    | 14        |
| JRE + JIT . . . . .              | 15        |
| Sicurezza . . . . .              | 16        |
| Esercizi . . . . .               | 17        |
| <b>Un po' di sintassi</b>        | <b>18</b> |
| Classi . . . . .                 | 19        |
| Attributi . . . . .              | 20        |
| Metodi . . . . .                 | 21        |
| Accesso a membri . . . . .       | 22        |
| <b>Incapsulazione</b>            | <b>23</b> |
| Il problema . . . . .            | 24        |
| La soluzione . . . . .           | 25        |
| <b>Costruttori</b>               | <b>26</b> |
| Sintassi . . . . .               | 27        |
| Attenzione . . . . .             | 28        |
| Costr. di default . . . . .      | 29        |
| <b>File sorgenti e pacchetti</b> | <b>30</b> |
| Sorgenti. . . . .                | 31        |
| Pacchetti . . . . .              | 32        |
| Enunciato package . . . . .      | 33        |

|                                    |           |
|------------------------------------|-----------|
| Enunciato import . . . . .         | 34        |
| <b>Cartelle e pacchetti</b>        | <b>35</b> |
| Organizz. consigliata . . . . .    | 36        |
| Ordine! . . . . .                  | 37        |
| Come? . . . . .                    | 38        |
| Fase di rilascio . . . . .         | 39        |
| Uso della documentazione . . . . . | 40        |
| Esercizi . . . . .                 | 41        |

## Panoramica della lezione

Introduzione a Java

Un esempio

JVM e JRE

Un po' di sintassi

Incapsulazione

Costruttori

File sorgenti e pacchetti

Cartelle e pacchetti

LP1 – Lezione 7

2 / 41

## Introduzione a Java

3 / 41

### Tecnologia Java

- **Un linguaggio di programmazione**, con sintassi simile a C++ e semantica simile a SmallTalk. È di solito menzionato (ma non serve solo) per produrre *applet*: applicazioni WEB che risiedono sul server ma sono eseguite dal browser WEB del cliente.  
Le *applicazioni* Java sono programmi autonomi che non richiedono un browser WEB per essere eseguiti. Tali programmi richiedono solo 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, ...

LP1 – Lezione 7

4 / 41

## I sorgenti

File: Saluti.java

```
public class Saluti {
 private String saluto;
 Saluti(String s) {
 saluto = s;
 }
 public void saluta(String chi) {
 System.out.println(saluto + " " + chi);
 }
}
```

File: ProvaSaluti.java

```
public class ProvaSaluti {
 public static void main(String[] args) {
 Saluti ciao = new Saluti("Ciao");
 Saluti arrivederci = new Saluti("Arrivederci");
 ciao.saluta("Alberto");
 ciao.saluta("Marcello");
 arrivederci.saluta("Stefania");
 }
}
```

LP1 – Lezione 7

6 / 41

## Compilazione ed esecuzione

Posto che i due file precedenti siano nella cartella corrente, il comando:

```
$ javac ProvaSaluti.java
```

produrrà il file oggetto ProvaSaluti.class, ma anche il file Saluti.class.

Una esecuzione attraverso l'interprete Java:

```
$ java ProvaSaluti
Ciao Alberto
Ciao Marcello
Arrivederci Stefania
$
```

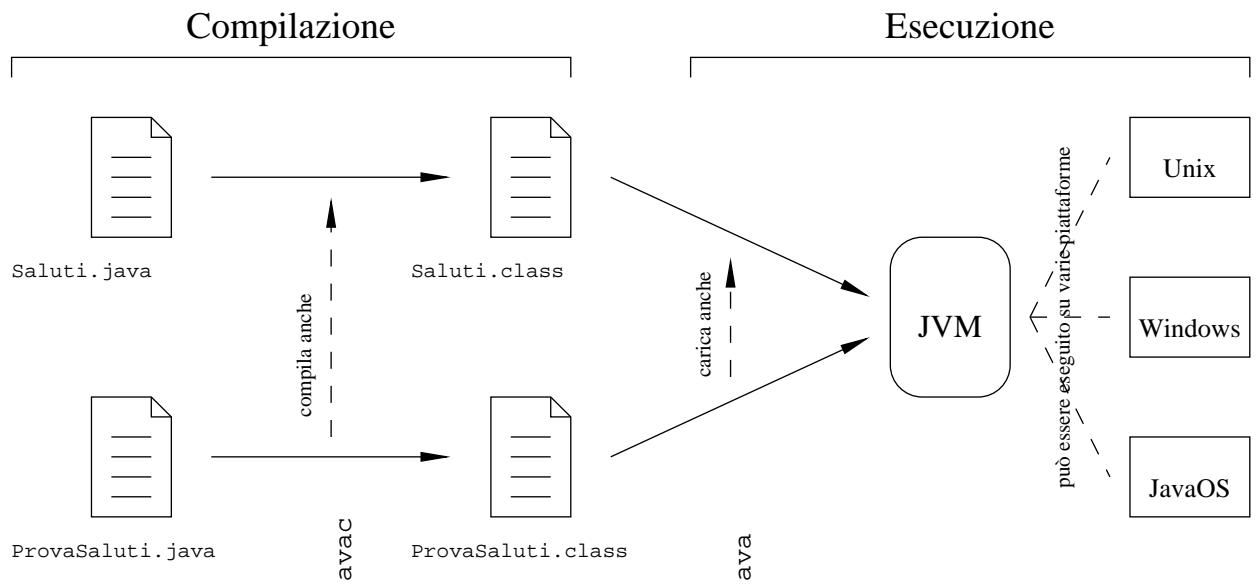
LP1 – Lezione 7

7 / 41

## 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.

## L'esempio esemplificato



## JVM

La Java Virtual Machine

*è una macchina immaginaria che è emulata dalla macchina reale. I programmi per la JVM sono contenuti in file .class, ciascuno dei quali contiene al più una classe pubblica*

Quindi, la JVM:

- fornisce le specifiche della piattaforma hardware;
- legge i codici *bytecode* compilati, che sono indipendenti dalla piattaforma;
- è realizzata in software o in hardware;
- è realizzata come strumento di sviluppo software oppure in un browser Web.

LP1 – Lezione 7

11 / 41

## Punti di vista

Dal punto di vista della realizzazione, la JVM specifica:

- l'insieme di istruzioni della CPU;
- l'insieme dei registri;
- il formato del file Class;
- lo stack di esecuzione;
- lo heap gestito dal garbage collector;
- l'area di memoria.

Dal punto di vista dell'utilizzatore:

- Il formato del programma compilato, eseguibile dalla JVM, consiste in *bytecode* molto compatti ed efficienti.
- La maggior parte del type-checking è svolto durante la compilazione.
- Ogni realizzazione della JVM deve essere capace di eseguire un qualunque programma conforme alle specifiche della JVM.

LP1 – Lezione 7

12 / 41

## Garbage Collector

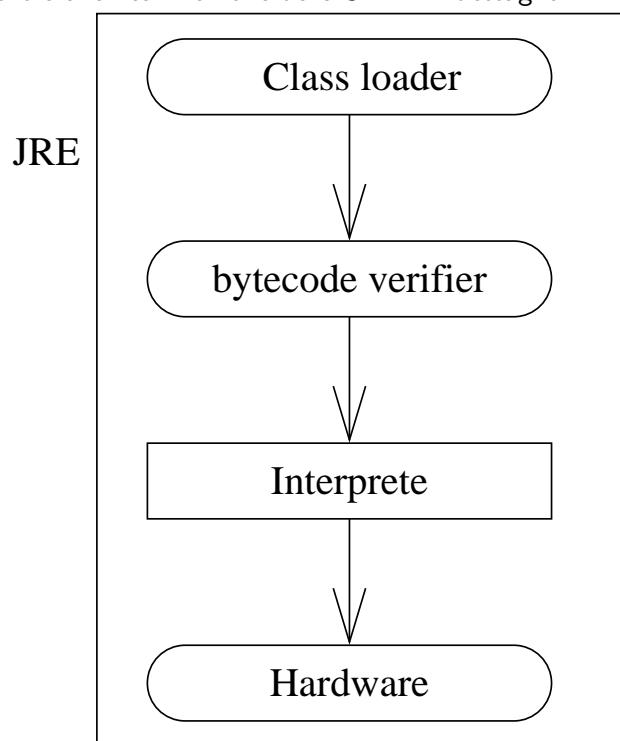
- La memoria allocata che non è più necessaria deve essere resa disponibile.
- In altri linguaggi la de-allocazione è responsabilità del programmatore.
- Java fornisce un thread a livello di sistema per tracciare l'allocazione di memoria.
- Il *Garbage Collector*:
  - ◆ ricerca e libera la memoria non più necessaria;
  - ◆ è eseguito automaticamente;
  - ◆ è dipendente dalle varie realizzazioni di JVM.

LP1 – Lezione 7

13 / 41

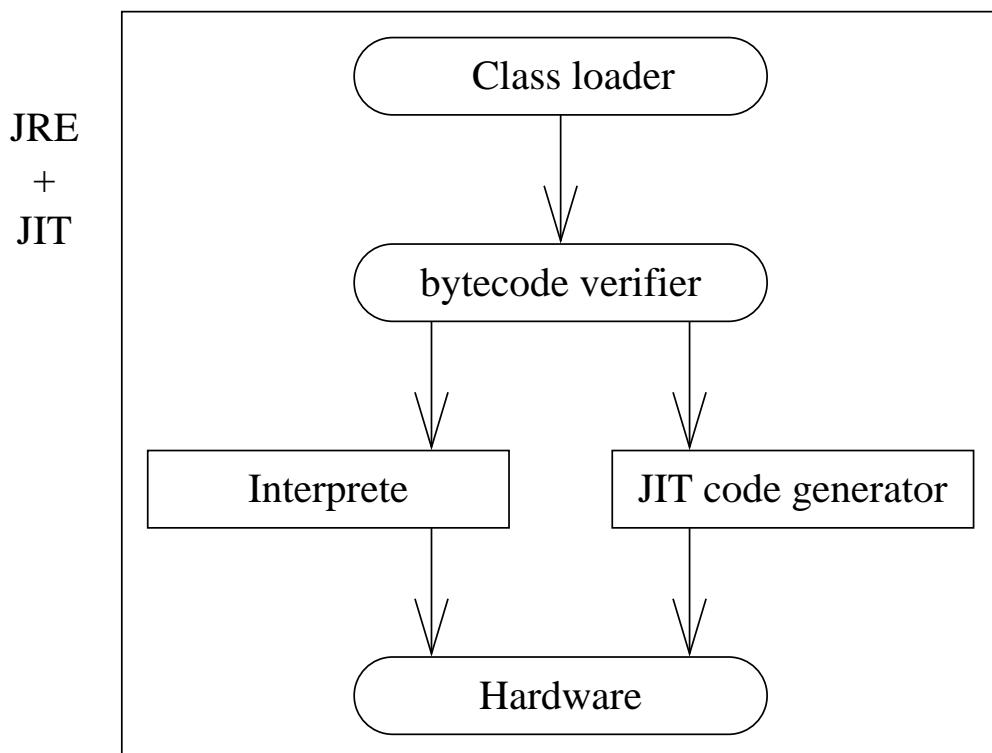
## JRE

Java Runtime Environment è una realizzazione della JVM. In dettaglio:



## JRE + JIT

Per migliorare la velocità di esecuzione:



LP1 – Lezione 7

15 / 41

## 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: ciò limita applicazioni "cavallo di Troia", poiché 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 (e.g. la conversione di interi in puntatori).

LP1 – Lezione 7

16 / 41

## Esercizi

1. Correggere gli errori nei frammenti di codice forniti.
2. Scrivere, compilare ed eseguire il programma precedente ProvaSaluti.

LP1 – Lezione 7

17 / 41

## Classi

La sintassi per la dichiarazione di una classe è:

```
<class_declarator> ::= <modifier> class <name> {
 <attribute_declarator>*
 <constructor_declarator>*
 <method_declarator>*
}
```

Esempio:

```
public class Automobile {
 private double maxPosti;
 public void setMaxPosti(double valore) {
 maxPosti = valore;
 }
}
```

LP1 – Lezione 7

19 / 41

## Attributi

La sintassi per la dichiarazione di un attributo è:

```
<attribute_declarator> ::=
 <modifier> <type> <name> [= <default_value>];

<type> ::= byte | short | int | long | char |
 float | double | boolean | <class>
```

Esempio:

```
public class Foo {
 public int x;
 public float y = 10000.0F;
 private String nome = "Marcello Sette";
}
```

LP1 – Lezione 7

20 / 41

## Metodi

La sintassi per la dichiarazione di metodi è:

```
<method_declarator> ::=
 <modifier> <return_type> <name> (<parameter>*) {
 <statement>*
 }
```

Esempio:

```
public class Cosa {
 private int x;
 public int getX () {
 return x;
 }
 public void setX (int nuovo_x) {
 x = nuovo_x;
 }
}
```

LP1 – Lezione 7

21 / 41

## Accesso a membri di oggetto

- La notazione "dot": <object>.<member>.
- È usata per l'accesso a membri **non privati** di una oggetto.
- Esempio:

```
public class TestCosa {
 public static void main (String[] args) {
 Cosa cc = new Cosa();

 cc.setX(47);
 System.out.println("cc.x vale " + cc.getX());
 }
}
```

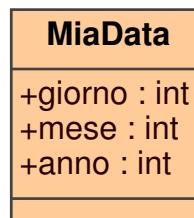
LP1 – Lezione 7

22 / 41

## Incapsulazione

23 / 41

### Il problema



Il codice cliente ha accesso diretto ai dati interni:

```
MiaData d = new MiaData();

d.giorno = 32;
// non valido

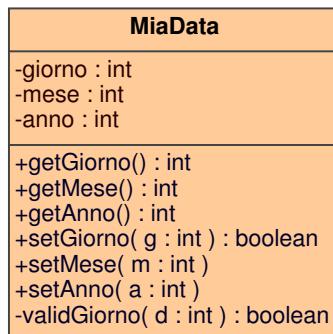
d.mese = 2; d.giorno = 30;
// plausibile ma sbagliato

d.giorno = d.giorno + 1;
// nessun controllo del limite superiore
```

LP1 – Lezione 7

24 / 41

## La soluzione



Il cliente deve usare i metodi per accedere ai dati interni:

```
MiaData d = new MiaData();

d.setGiorno(32);
// non valido, ma ritorna falso

d.setMese(2);
d.setGiorno(30);
// plausibile ma sbagliato; setGiorno ritorna false

d.setGiorno(d.getGiorno() + 1);
// restituisce false se si eccede il limite
```

LP1 – Lezione 7

25 / 41

## Costruttori

26 / 41

### Sintassi

La sintassi per la dichiarazione di un costruttore è:

```
<constructor_declarator> ::=
 <modifier> <class_name> (<parameter>*) {
 <statement>*
 }
```

Esempio:

```
public class Cosa {
 private int x;
 public Cosa () {
 x = 47;
 }
 public Cosa (int nuova_x) {
 x = nuova_x;
 }
}
```

LP1 – Lezione 7

27 / 41

## Attenzione

- i costruttori NON sono metodi;
- i costruttori NON sono ereditati;
- un costruttore NON ha valore di ritorno;
- gli unici modificatori validi per il costruttore sono: public, protected e private.

LP1 – Lezione 7

28 / 41

## 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.
- Il costruttore di default non ha corpo.
- Il costruttore di default permette di creare istanze di classi (con l'espressione new Xxx()) senza dover scrivere un costruttore.

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

LP1 – Lezione 7

29 / 41

## File sorgenti e pacchetti

30 / 41

### Sorgenti

```
<source_file> ::=
[<package_declaraction>]
<import_declaraction>*
<class_declaraction>*
```

- l'ordine degli elementi è importante;
- il nome del file sorgente deve essere lo stesso dell'unica classe pubblica contenuta nel file; se il file sorgente non contiene classi pubbliche allora il nome del file può essere qualunque.  
Esempio: il file RapportoCapacitaVeicolo.java

```
package trasporti.rapporti.Web;

import trasporti.dominio.*;
import java.util.List;
import java.io.*;

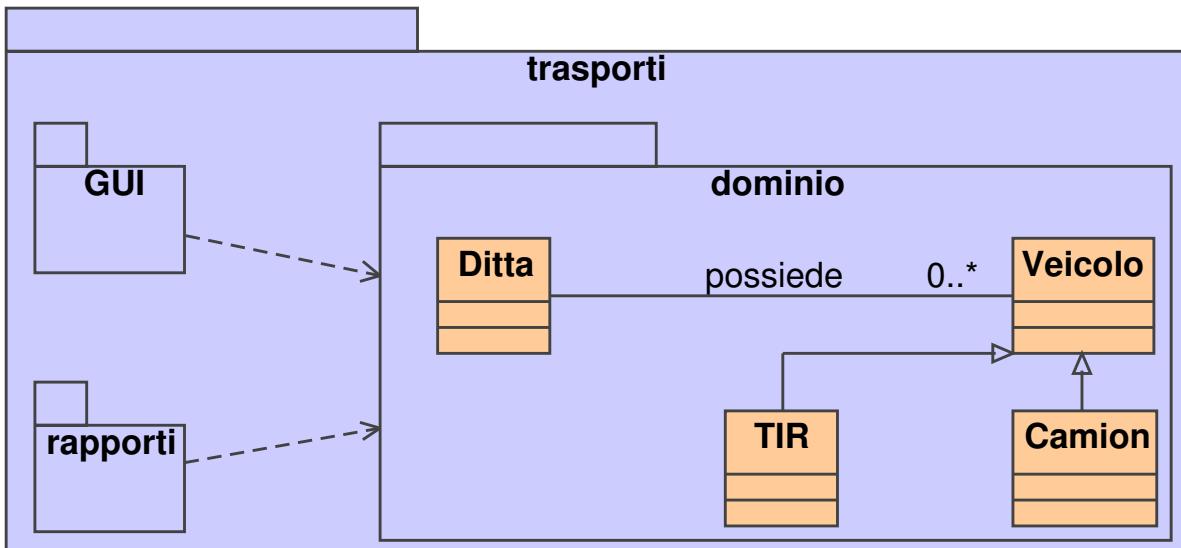
public class RapportoCapacitaVeicolo {
 private List veicoli;
 public void generaRapporto (Writer output) {
 ...
 }
}
```

LP1 – Lezione 7

31 / 41

## Pacchetti

- Aiutano a gestire grandi sistemi software.
- Possono contenere classi e sotto-pacchetti.



## Enunciato package

La sintassi dell'enunciato package è:

```
<package_declaraction> ::=
 package <top_pkg_name>[.<sub_pkg_name>]*;
```

ove:

- la dichiarazione specifica che il contenuto del file appartiene al pacchetto dichiarato;
- è permessa una sola dichiarazione di pacchetto per file sorgente;
- si deve specificare la dichiarazione all'inizio del file sorgente;
- se non è dichiarato il pacchetto, allora il contenuto del file appartiene al pacchetto di default;
- i nomi di pacchetto devono essere gerarchici e separati da punti;
- un nome di pacchetto corrisponde di solito ad un nome di cartella;
- in genere i nomi di pacchetto sono scritti in lettera minuscola, mentre i nomi di classe sono in lettera minuscola ma ogni loro parola comincia per lettera maiuscola.

## Enunciato import

La sintassi dell'enunciato import è:

```
<import_declar> ::=
 import <top_pkg_name>[.<sub_pkg_name>]*.<classes>;

<classes> ::= <class_name> | *
```

- Precede ogni dichiarazione di classe.
- Dice al compilatore dove trovare le classi da usare.
- Importa solo lo spazio dei nomi, non le classi. Serve cioè solo ad abbreviare la scrittura del codice, permettendo di citare solo il nome di una classe e non obbligatoriamente tutto il percorso per raggiungerla.

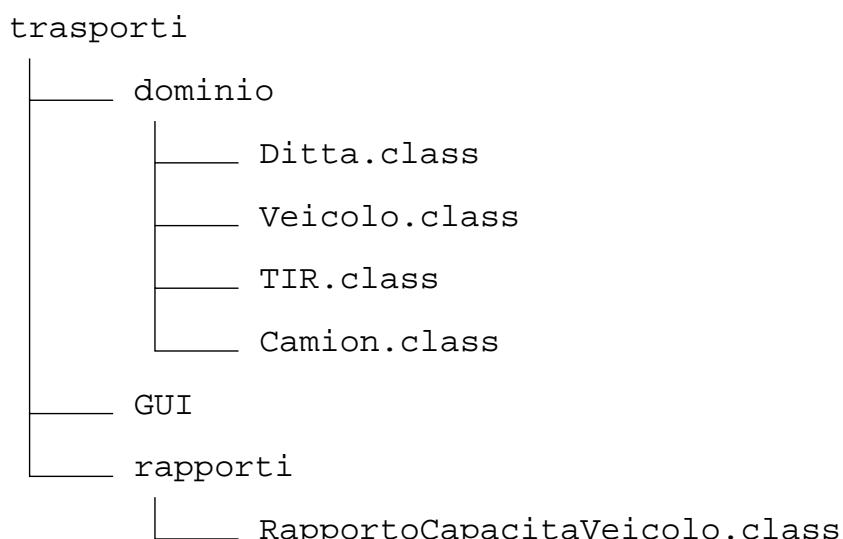
Esempio:

```
import trasporti.dominio.*;
import java.util.List;
import java.io.*;
```

## Cartelle e pacchetti

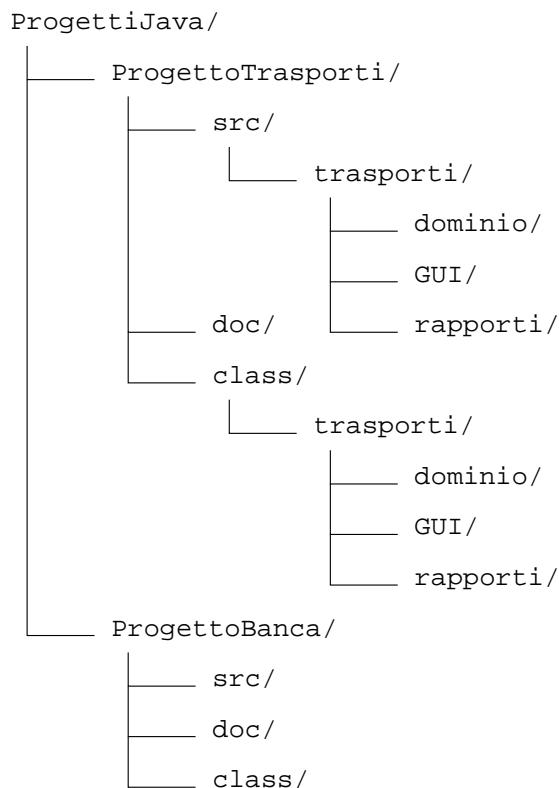
### Organizzazione consigliata

- Pacchetti contenuti in alberi di cartelle.
- Nomi di cartelle uguali a nomi di pacchetto.
- Nell'esempio precedente:



## Ordine!

È buona norma di programmazione quella di separare, nello sviluppo di un progetto, i file sorgente dai file compilati. Un metodo possibile è:



LP1 – Lezione 7

37 / 41

## Come?

- Normalmente il compilatore pone i file .class nella stessa cartella dei file .java.
- I file .class possono essere re-indirizzati in un'altra cartella usando l'opzione -d del comando javac.
- Se si sta compilando un insieme di file all'interno della gerarchia dei pacchetti (cioè non nella cartella principale dei sorgenti), allora bisogna anche specificare l'opzione -sourcepath.

Per esempio:

```
$ cd ProgettiJava/ProgettoTrasporti/src/trasporti/dominio
$ javac -sourcepath ProgettiJava/ProgettoTrasporti/src \
 -d ProgettiJava/ProgettoTrasporti/class *.java
```

LP1 – Lezione 7

38 / 41

## Fase di rilascio

Una applicazione può essere rilasciata su una macchina cliente in più modi:

1. Se si utilizza un file JAR, quel file deve essere copiato nella cartella delle “estensioni di libreria” .

Per esempio:

/usr/jdk1.4/jre/lib/ext/

Ambiente Linux

C:\jdk1.4\jre\lib\ext\

Ambiente Windows

2. Se l'applicazione è rilasciata come una gerarchia di file .class, allora bisogna porre tutta la gerarchia all'interno della cartella delle “classi JRE”. Per esempio:

/usr/jdk1.4/jre/classes/

Ambiente Linux

C:\jdk1.4\jre\classes\

Ambiente Windows

LP1 – Lezione 7

39 / 41

## Uso della documentazione

- Dimostrazione sull'utilizzo.
- Descrizione della gerarchia delle classi della API.

LP1 – Lezione 7

40 / 41

## Esercizi

1. Uso della documentazione.
2. Realizzazione della encapsulazione.
3. Creazione di un semplice pacchetto.

LP1 – Lezione 7

41 / 41

---

# Linguaggi di Programmazione I – Lezione 1

Prof. P. A. Bonatti  
<mailto://bonatti@na.infn.it>  
<http://people.na.infn.it/~bonatti>

Si ringrazia il Prof. Marcello Sette per il materiale didattico

16 marzo 2020



# Panoramica della lezione

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Paradigmi computazionali



## Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Esame

Altre informazioni

Linguaggi (di  
programmazione)

Macchine astratte

Paradigmi  
computazionali

# Introduzione al corso



# Obiettivi specifici

[Introduzione al corso](#)

**Obiettivi specifici**

[Obiettivi generali](#)

[Scheduling](#)

[Esame](#)

[Altre informazioni](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

- Mettere lo studente in grado di imparare velocemente un nuovo linguaggio di programmazione
  - ◆ mediante astrazione delle *caratteristiche costituenti* dei linguaggi
  - ◆ cambiano più di rado di quanto vengano introdotti nuovi linguaggi
- Spiegare le differenze tra i vari paradigmi di programmazione - imperativo, ad oggetti, funzionale, logico - e sul loro impatto sullo stile di soluzione dei problemi.
- Mostrare cenni sui criteri di progetto e le tecniche d'implementazione dei linguaggi di programmazione
- Esempi focalizzati su Java, ML, Prolog.



# Obiettivi generali

- Migliorare l'abilità nel risolvere i problemi usando meglio i linguaggi di programmazione.

[Introduzione al corso](#)

[Obiettivi specifici](#)

**Obiettivi generali**

[Scheduling](#)

[Esame](#)

[Altre informazioni](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)



# Obiettivi generali

- Migliorare l'abilità nel risolvere i problemi usando meglio i linguaggi di programmazione.
- Imparare a scegliere più intelligentemente, in dipendenza del problema, il linguaggio di programmazione.

[Introduzione al corso](#)

[Obiettivi specifici](#)

**Obiettivi generali**

[Scheduling](#)

[Esame](#)

[Altre informazioni](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)



# Obiettivi generali

- Migliorare l'abilità nel risolvere i problemi usando meglio i linguaggi di programmazione.
- Imparare a scegliere più intelligentemente, in dipendenza del problema, il linguaggio di programmazione.
- Aumentare la capacità di imparare linguaggi di programmazione (che sono TANTI!).

[Introduzione al corso](#)

[Obiettivi specifici](#)

**Obiettivi generali**

[Scheduling](#)

[Esame](#)

[Altre informazioni](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Scheduling

[Introduzione al corso](#)

Obiettivi specifici

Obiettivi generali

**Scheduling**

Esame

Altre informazioni

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

## Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.



# Scheduling

[Introduzione al corso](#)

Obiettivi specifici

Obiettivi generali

**Scheduling**

Esame

Altre informazioni

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

## ■ Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.

## ■ Parte seconda (modello object-oriented, Java)

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



# Scheduling

[Introduzione al corso](#)

Obiettivi specifici

Obiettivi generali

**Scheduling**

Esame

Altre informazioni

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

## ■ Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.

## ■ Parte seconda (modello object-oriented, Java)

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

## ■ Parte terza (Linguaggi funzionali, ML)

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



# Scheduling

[Introduzione al corso](#)

Obiettivi specifici

Obiettivi generali

**Scheduling**

Esame

Altre informazioni

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

## ■ Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.

## ■ Parte seconda (modello object-oriented, Java)

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

## ■ Parte terza (Linguaggi funzionali, ML)

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

## ■ Parte quarta (Linguaggi logici, Prolog)

1. Costrutti fondamentali: termini, predicati e regole.
2. Tecniche di programmazione avanzate: invertibilità e nondeterminismo.



## Esame

L'esame consisterà in due scritti:

1. su Java
2. sul resto del corso

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

**Esame**

Altre informazioni

Linguaggi (di  
programmazione)

Macchine astratte

Paradigmi  
computazionali



## Esame

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

**Esame**

Altre informazioni

Linguaggi (di  
programmazione)

Macchine astratte

Paradigmi  
computazionali

L'esame consisterà in due scritti:

1. su Java
2. sul resto del corso

Solo in casi dubbi per il docente, un orale. Inoltre:

- E' possibile mantenere il voto di una prova scritta per la durata di un anno accademico
- L'ultima prova sostenuta cancella le precedenti (anche se è andata peggio)



## Altre informazioni

### ■ *Libri di testo:*

- ◆ Dershem - Jipping. *Programming languages: structures and models.*
- ◆ Wampler. *The essence on object oriented programming with Java and UML.*
- ◆ Fowler. *UML distilled.*
- ◆ Eckel. *Thinking in Java.*
- ◆ Gabbrielli - Martini. *Linguaggi di programmazione: Principi e paradigmi*
- ◆ Altro materiale fornito dal docente

[Introduzione al corso](#)

[Obiettivi specifici](#)

[Obiettivi generali](#)

[Scheduling](#)

[Esame](#)

**Altre informazioni**

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Altre informazioni

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Esame

**Altre informazioni**

Linguaggi (di  
programmazione)

Macchine astratte

Paradigmi  
computazionali

## ■ *Libri di testo:*

- ◆ Dershem - Jipping. *Programming languages: structures and models.*
- ◆ Wampler. *The essence on object oriented programming with Java and UML.*
- ◆ Fowler. *UML distilled.*
- ◆ Eckel. *Thinking in Java.*
- ◆ Gabbrielli - Martini. *Linguaggi di programmazione: Principi e paradigmi*
- ◆ Altro materiale fornito dal docente

## ■ *Materiali:* La maggior parte si trovano su

<http://wpage.unina.it/pieroandrea.bonatti/didattica/#LI>

■ il nuovo materiale sarà invece pubblicato su [docenti.unina.it](http://docenti.unina.it)

■ *Ricevimento studenti:* per appuntamento con email al docente



## Introduzione al corso

### Linguaggi (di programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

Linguaggi completi

## Macchine astratte

### Paradigmi computazionali

# Linguaggi (di programmazione)



# Sono ~ 40?

Introduzione al corso

Linguaggi (di programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

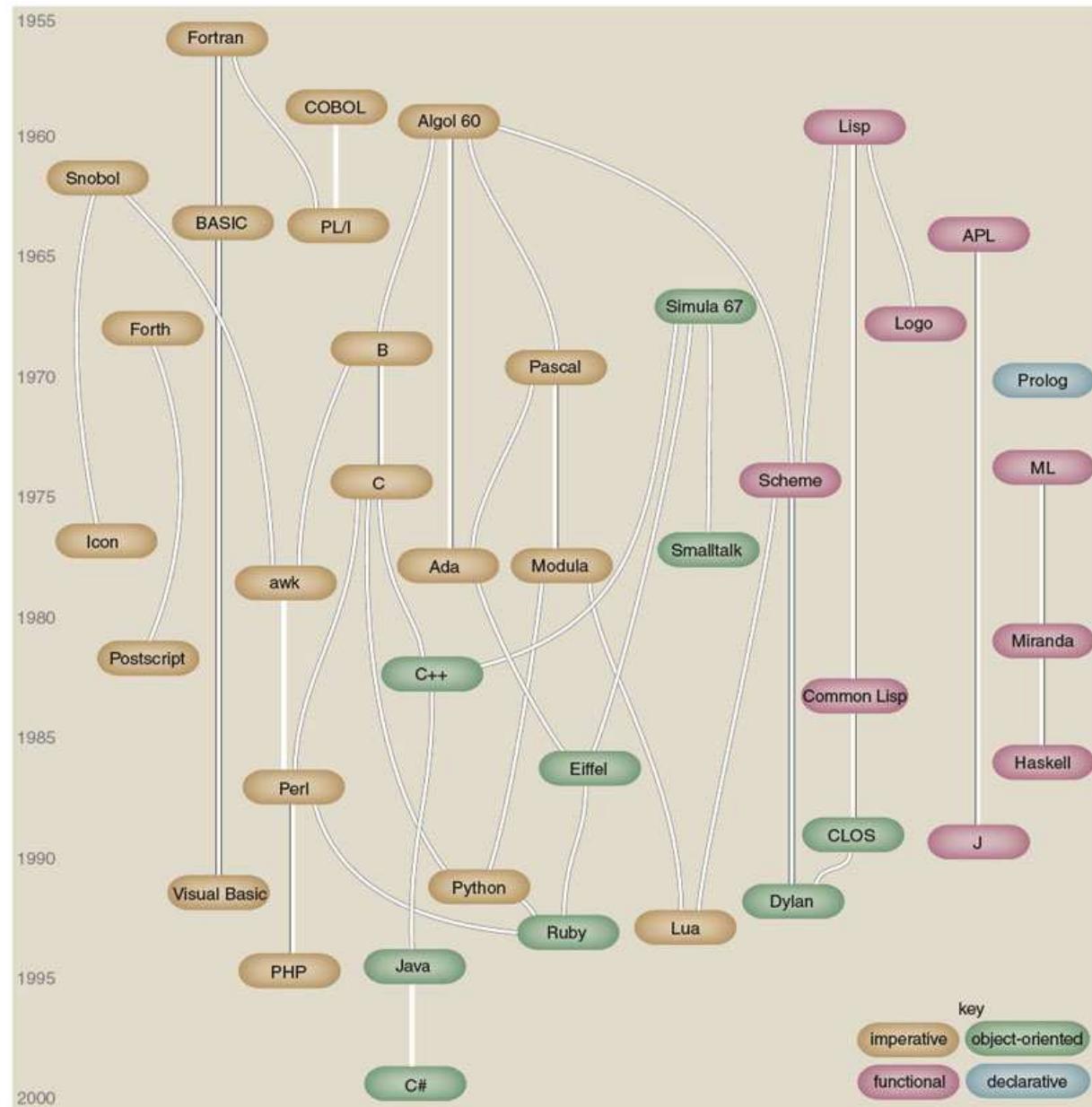
. . . concetti  
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi  
computazionali





# Sono ~ 80?

Introduzione al corso

Linguaggi (di programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi computazionali

## Mother Tongues

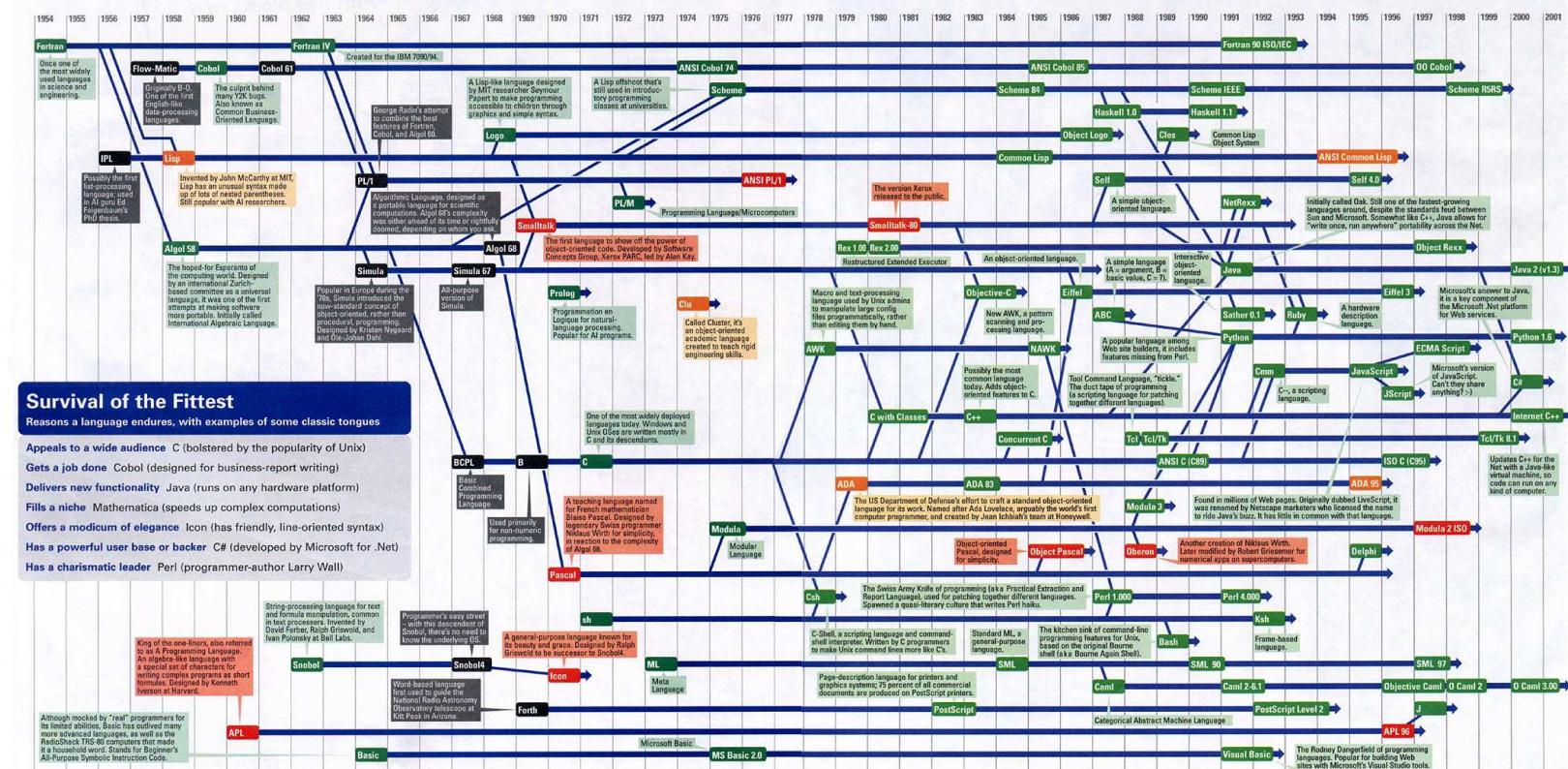
Tracing the roots of computer languages through the ages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java, and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers—electronic lexicographers, if you will—aim to save, or at least document, the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at [www.informatik.uni-freiburg.de/Java/misc/lang\\_list.html](http://www.informatik.uni-freiburg.de/Java/misc/lang_list.html)—Michael Menduno

| Key                                                                    | Year Introduced                                        |
|------------------------------------------------------------------------|--------------------------------------------------------|
| Once one of the most widely used languages in science and engineering. | Active: thousands of users                             |
| Originally B-O, designed for the IBM 7090/94.                          | Protected: taught at universities; compilers available |
| Also known as Common Business-Oriented Language.                       | Endangered: usage dropping off                         |
| Flow-Matic                                                             | Extinct: no known active users or up-to-date compilers |
| IPL                                                                    | Lineage continues                                      |





# Quanti sono?

## Introduzione al corso

## Linguaggi (di programmazione)

Sono  $\sim 40$ ?

Sono  $\sim$  80?

## Quanti sono?

## Quanti sono?

Breve storia

Storia dei ...

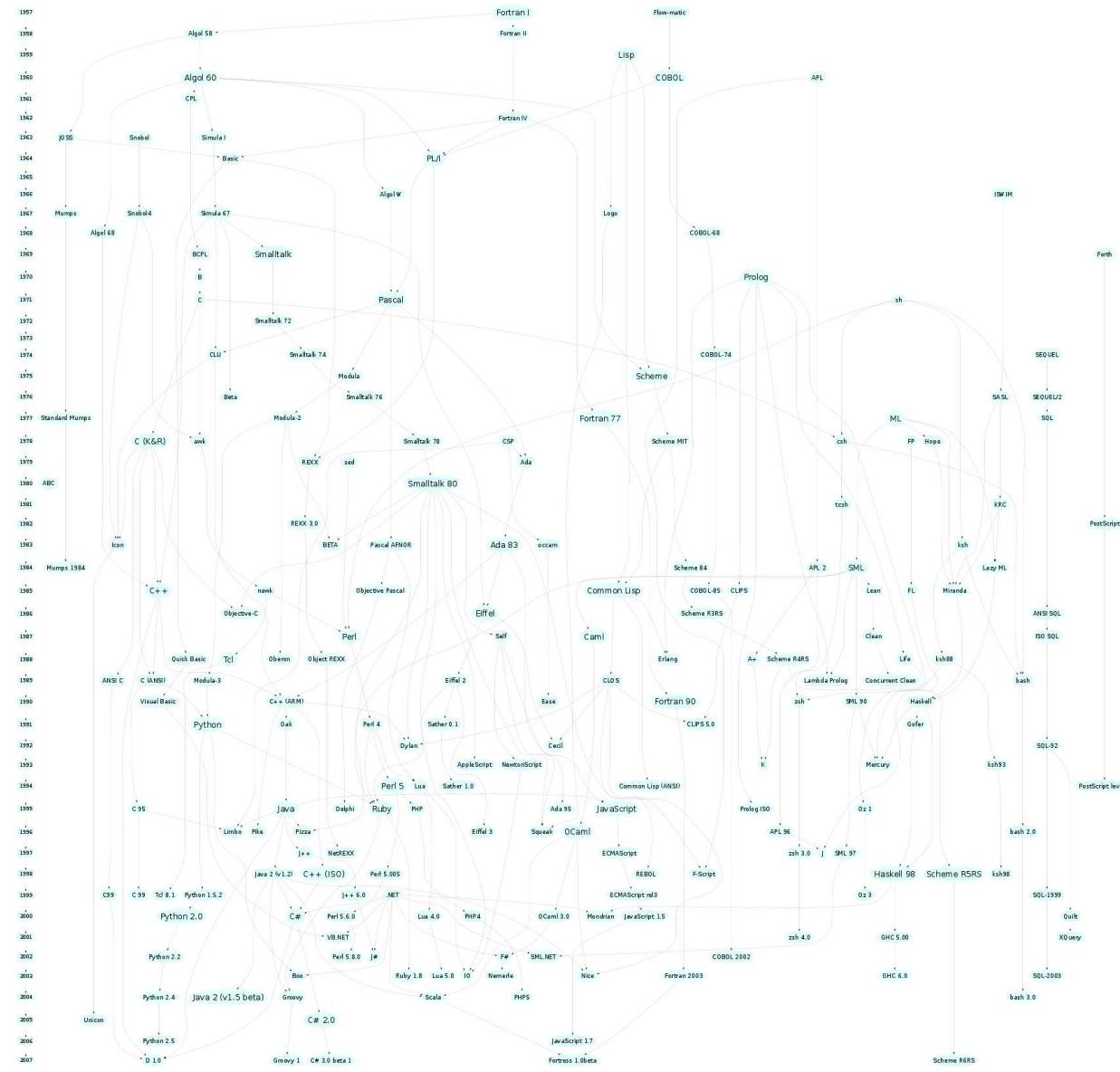
... concetti  
introdotti

## Terminologia

## Linguaggi completi

## Macchine astratte

# Paradigmi computazionali





# Quanti sono?

- C'è chi dice addirittura qualche migliaio
  - ◆ Come orientarsi?
  - ◆ Come usarli *bene*?
  - ◆ Come apprendere in fretta quelli nuovi?
- Occorre comprensione astratta delle caratteristiche dei linguaggi per coglierne somiglianze/differenze
- e per comprendere lo scopo di ciascun costrutto (ovvero i principi del *language design*)

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

**Quanti sono?**

Breve storia

Storia dei . . .

. . . concetti introdotti

Terminologia

Linguaggi completi

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Breve storia

1954

FORTRAN (FORmula TRANslation)

Introduzione al corso

Linguaggi (di  
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti  
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi  
computazionali



## Breve storia

Introduzione al corso

Linguaggi (di  
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

**Breve storia**

Storia dei . . .

. . . concetti  
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi  
computazionali

|      |                                                                                                                                                                                                     |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1954 | FORTRAN (FORmula TRANslation)                                                                                                                                                                       |
| 1960 | COBOL (Common Business Oriented Language)<br>ALGOL 60 (Algorithmic Oriented Language)<br>PL/1 (Programming Language 1)<br>Simula 67<br>ALGOL 68<br>PASCAL<br>LISP (LISt Processing)<br>APL<br>BASIC |



## Breve storia

Introduzione al corso

Linguaggi (di  
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

**Breve storia**

Storia dei . . .

. . . concetti  
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi  
computazionali

|         |                                                                                                                                                                                                     |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1954    | FORTRAN (FORmula TRANslation)                                                                                                                                                                       |
| 1960    | COBOL (Common Business Oriented Language)<br>ALGOL 60 (Algorithmic Oriented Language)<br>PL/1 (Programming Language 1)<br>Simula 67<br>ALGOL 68<br>PASCAL<br>LISP (LISt Processing)<br>APL<br>BASIC |
| 1970/80 | PROLOG<br>SMALLTALK<br>C<br>MODULA/2<br>ADA                                                                                                                                                         |



# Storia dei concetti introdotti dai progenitori dell'oggi

**Fortran:** nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.



# Storia dei concetti introdotti dai progenitori dell'oggi

**Fortran:** nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

**Cobol:** indipendenza dalla macchina e statement “English like”. Orientato ai database. Introduce il record.



# Storia dei concetti introdotti dai progenitori dell'oggi

**Fortran:** nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

**Cobol:** indipendenza dalla macchina e statement “English like”. Orientato ai database. Introduce il record.

**Algol60:** indipendenza dalla macchina e definizione mediante

grammatica(bakus-naur form), strutture a blocco, supporto generale dell'iterazione e ricorsione.



# Storia dei concetti introdotti dai progenitori dell'oggi

**Fortran:** nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

**Cobol:** indipendenza dalla macchina e statement “English like”. Orientato ai database. Introduce il record.

**Algol60:** indipendenza dalla macchina e definizione mediante

grammatica(bakus-naur form), strutture a blocco, supporto generale dell'iterazione e ricorsione.

**Lisp:** primo vero linguaggio di manipolazione simbolica, paradigma funzionale, non c'è lo statement di assegnazione, e quindi concettualmente non c'è “il valore” ovvero l'idea di cambiare lo stato della memoria. Non c'è differenza concettuale fra funzione e dato: dipende dall'uso. Prima versione essenzialmente non tipata.



# Storia dei concetti introdotti dai progenitori dell'oggi

**Prolog:** primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.



# Storia dei concetti introdotti dai progenitori dell'oggi

**Prolog:** primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

**Simula 67:** classe come encapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di

dato astratto implementati in Ada e Modula2, e del concetto di classe di Smalltalk e C++.



# Storia dei concetti introdotti dai progenitori dell'oggi

**Prolog:** primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

**Simula 67:** classe come encapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di

dato astratto implementati in Ada e Modula2, e del concetto di classe di Smalltalk e C++.

**PL/1:** abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; “multitasking”, cioè specificazione di tasks che possono essere eseguiti in concorrenza.



# Storia dei concetti introdotti dai progenitori dell'oggi

**Prolog:** primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

**Simula 67:** classe come encapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di

dato astratto implementati in Ada e Modula2, e del concetto di classe di Smalltalk e C++.

**PL/1:** abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; “multitasking”, cioè specificazione di tasks che possono essere eseguiti in concorrenza.

**Pascal:** programmazione strutturata, tipi di dato definiti da utente, ricchezza di strutture dati. Ma ancora niente encapsulation; si dovrà aspettare Modula.



# Terminologia

- **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.

Introduzione al corso

Linguaggi (di  
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi  
computazionali



# Terminologia

- **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; non si deve intendere come un singolo oggetto, ma come una *architettura di elaborazione*.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti introdotti

**Terminologia**

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Terminologia

- **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; non si deve intendere come un singolo oggetto, ma come una *architettura di elaborazione*.
- **Programma:** è l'espressione codificata di un processo.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti introdotti

**Terminologia**

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono programmare *qualunque funzione calcolabile*
  - ◆ detti anche *general purpose*
  - ◆ tecnicamente: devono poter simulare qualunque *macchina di Turing*.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

**Linguaggi completi**

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono programmare *qualunque funzione calcolabile*
  - ◆ detti anche *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

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono programmare *qualunque funzione calcolabile*
  - ◆ detti anche *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
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile).

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono programmare *qualunque funzione calcolabile*
  - ◆ detti anche *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
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile). È spesso immerso in linguaggi completi.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



# Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono programmare *qualunque funzione calcolabile*
  - ◆ detti anche *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
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile). È spesso immerso in linguaggi completi.
- HTML non è un linguaggio completo (idem).
- Per mostrare la completezza di un linguaggio: usarlo per simulare arbitrarie macchine di Turing

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Introduzione al corso

Linguaggi (di  
programmazione)

**Macchine astratte**

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed . . .

Compilatore

Proprietà dei . . .

Criteri di scelta . . .

Paradigmi  
computazionali

# Macchine astratte



## Definizione

Dato un linguaggio di programmazione  $L$ , una macchina astratta per  $L$  (in simboli,  $M_L$ ) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in  $L$ .

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali



# Definizione

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

**Definizione**

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

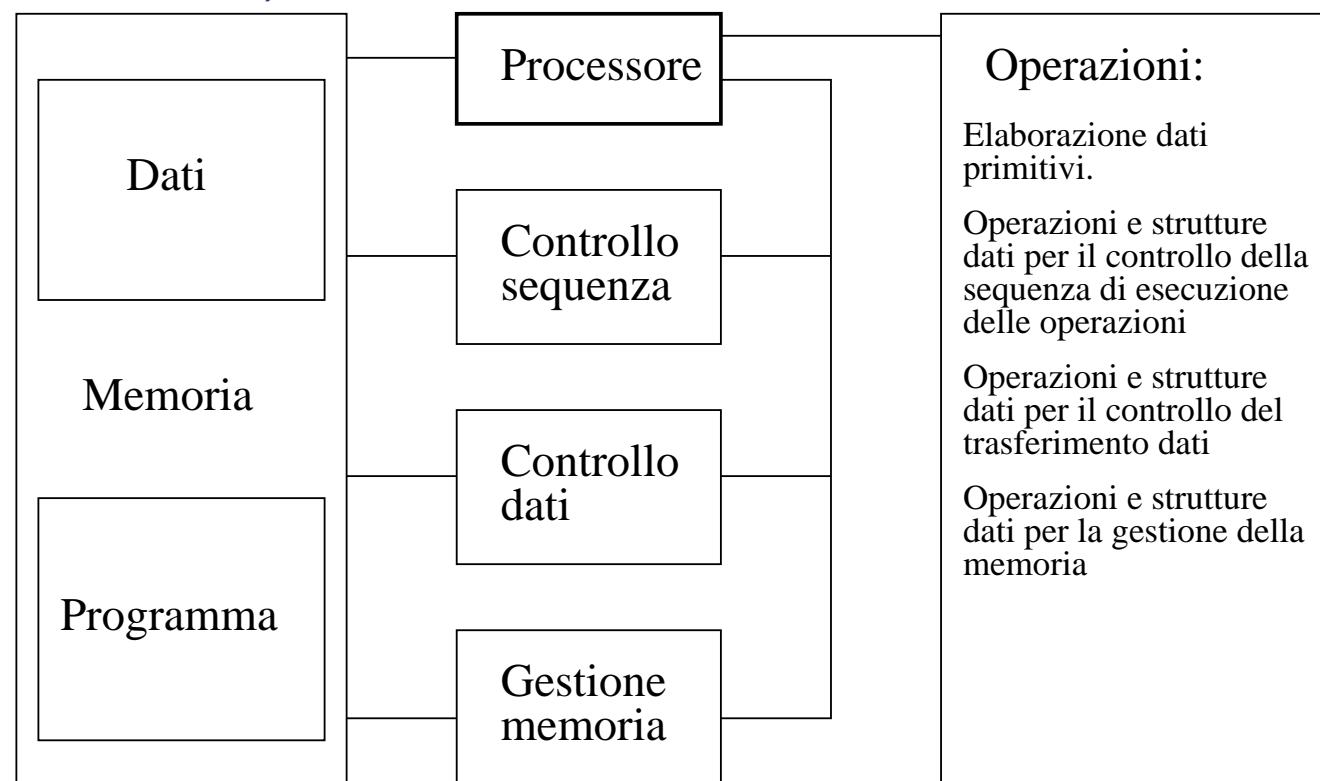
Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

Dato un linguaggio di programmazione  $L$ , una macchina astratta per  $L$  (in simboli,  $M_L$ ) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in  $L$ .

La struttura di una macchina astratta è (essenzialmente memoria e processore):





# Esempio

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

**Esempio**

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

## Macchina E–Business (commercio online)

### Macchina Web Service

#### Macchina Web

#### Macchina linguaggio di programmazione

#### Macchina intermedia (java bytecode)

#### Macchina Sistema Operativo

#### Macchina firmware

#### Macchina hardware



# Processore della macchina astratta

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

**Processore M.A.**

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

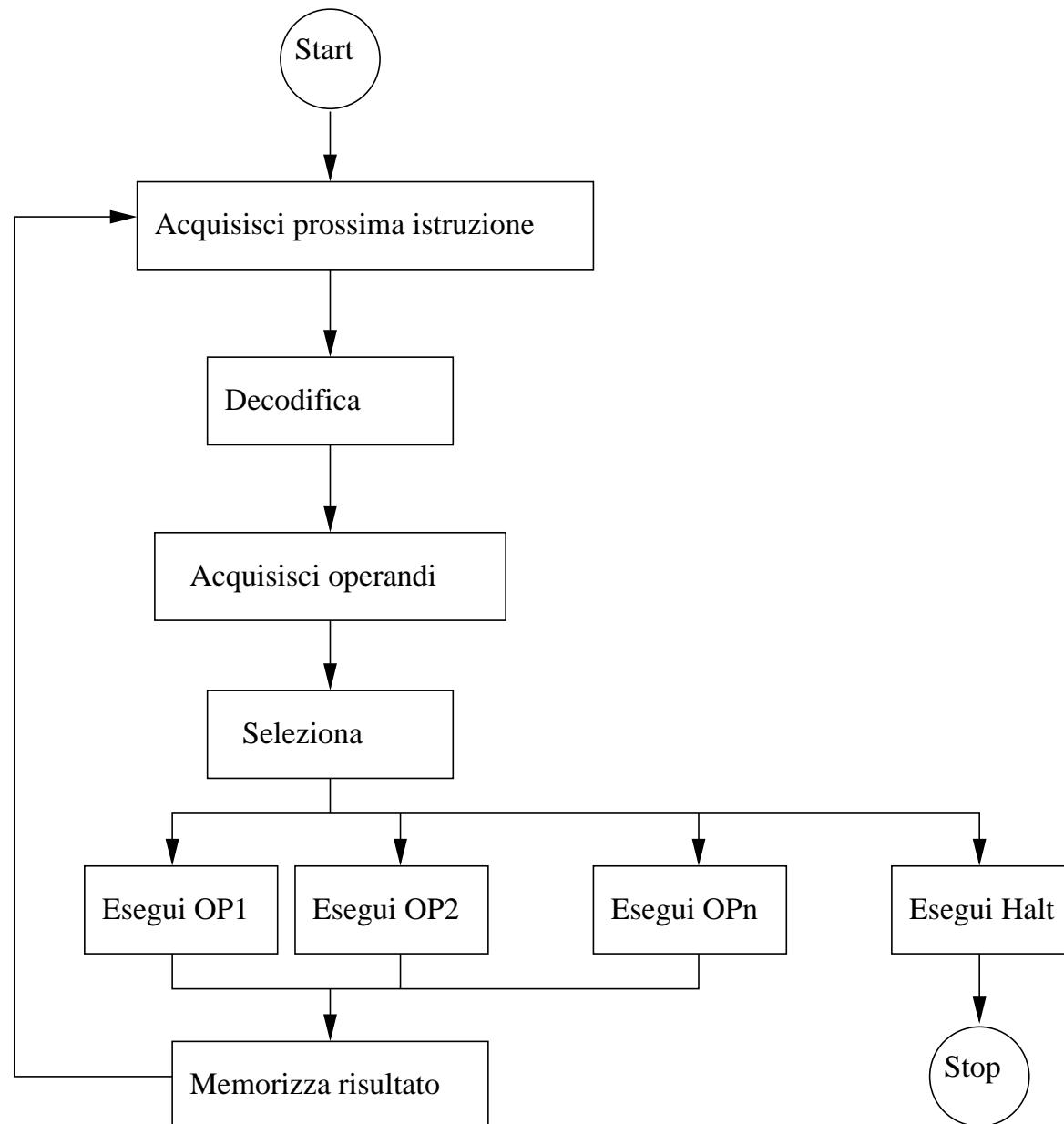
Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali





# Tecnologie di realizzazione di macchina astratta

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

**Realizzazione M.A.**

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

**Hardware:** Si era pensato per Lisp e Prolog



# Tecnologie di realizzazione di macchina astratta

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

**Realizzazione M.A.**

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

**Hardware:** Si era pensato per Lisp e Prolog

**Firmware:** Soluzione più flessibile e semplice / economicità di progetto



# Tecnologie di realizzazione di macchina astratta

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

**Realizzazione M.A.**

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

**Hardware:** Si era pensato per Lisp e Prolog

**Firmware:** Soluzione più flessibile e semplice / economicità di progetto

**Software:** Ad es. macchina astratta Java, o Warren Abstract Machine (Prolog)



# Traduttori Linguaggio → Macchina astratta

- **Interpreti:** traducono ed eseguono un costrutto alla volta.  
PRO: debug - fase di sviluppo: interazione più snella.

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali



# Traduttori Linguaggio → Macchina astratta

- **Interpreti:** traducono ed eseguono un costrutto alla volta.  
PRO: debug - fase di sviluppo: interazione più snella.
- **Compilatori:** prima traducono l'intero programma; poi la traduzione può essere eseguita (anche più volte).  
PRO: velocità di esecuzione finale - fase di rilascio.  
PRO: più controlli e in anticipo

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali



# Interpretazione pura

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

**Interpretazione pura**

Compilazione 1

Compilazione 2

Compilazione 3

SRT

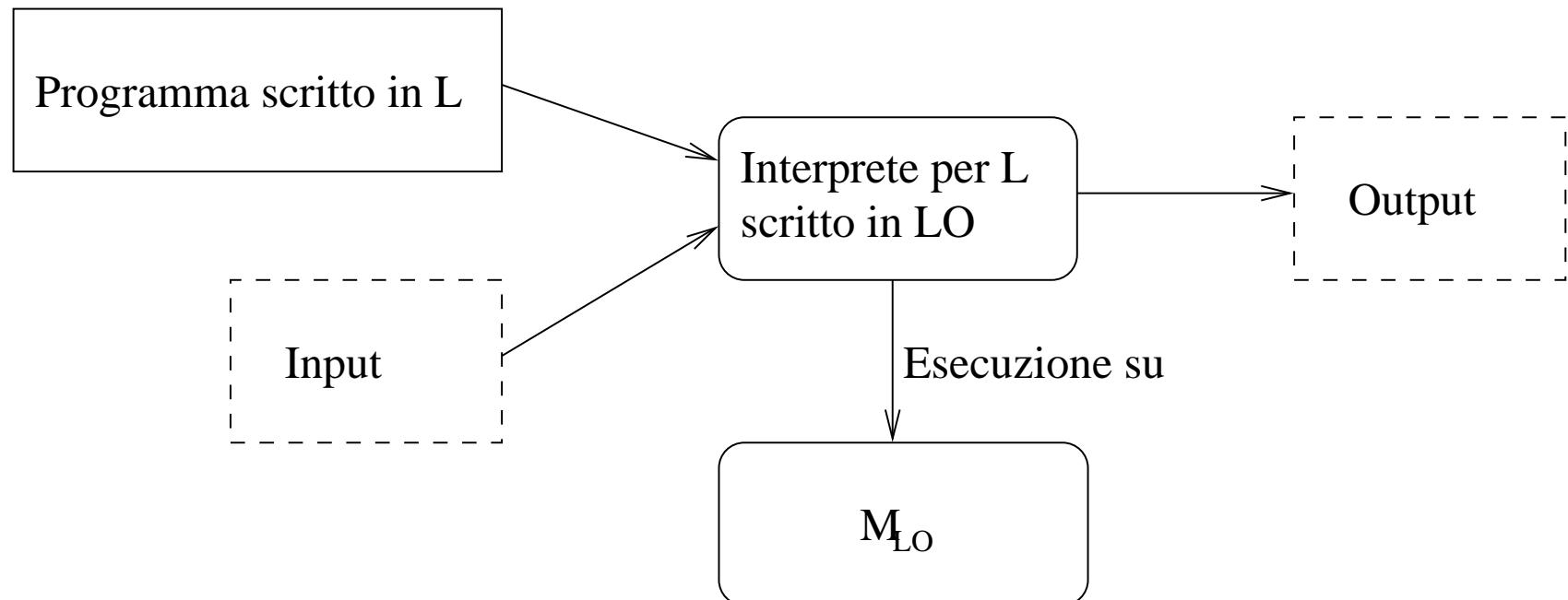
Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali





# Compilazione pura (caso semplice)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

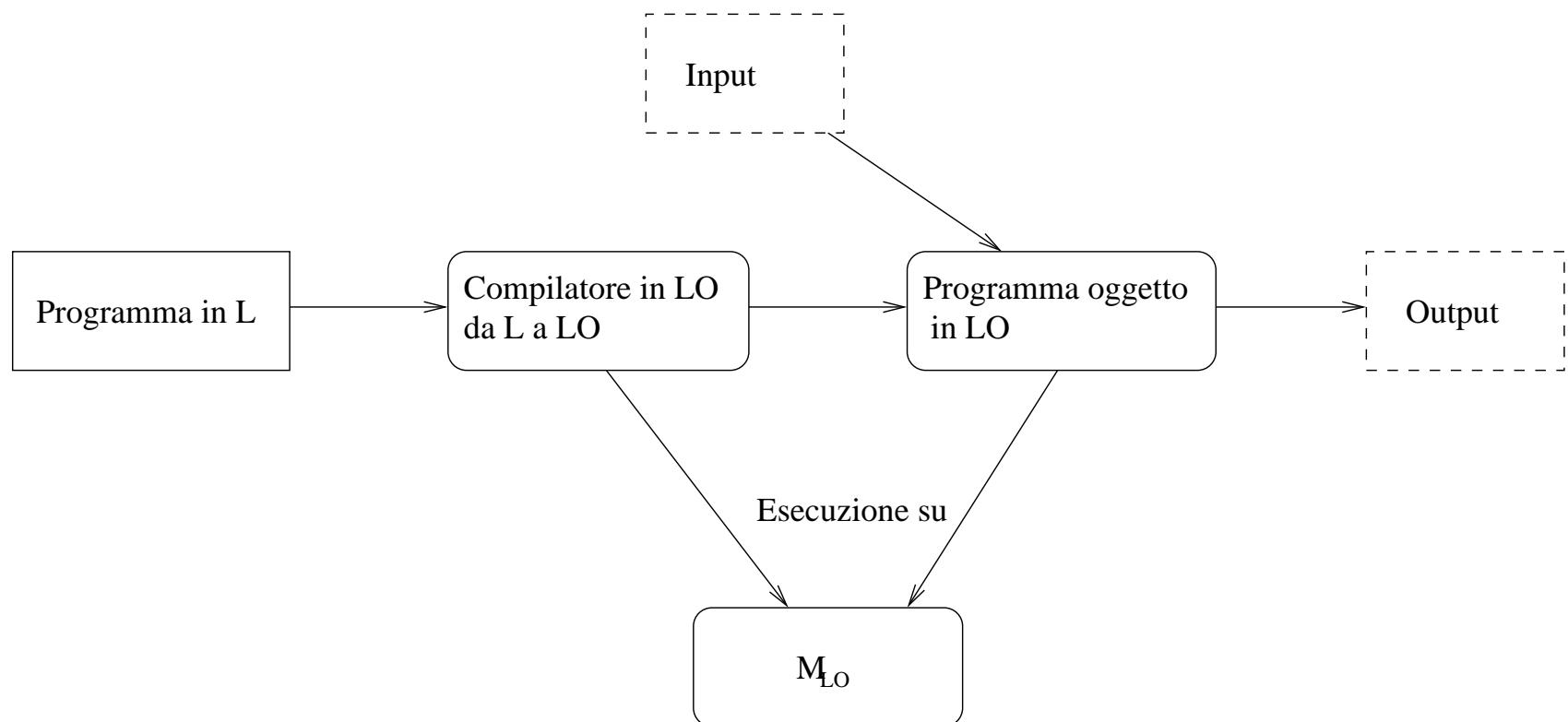
Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali





# Compilazione pura (caso più generale)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

**Compilazione 2**

Compilazione 3

SRT

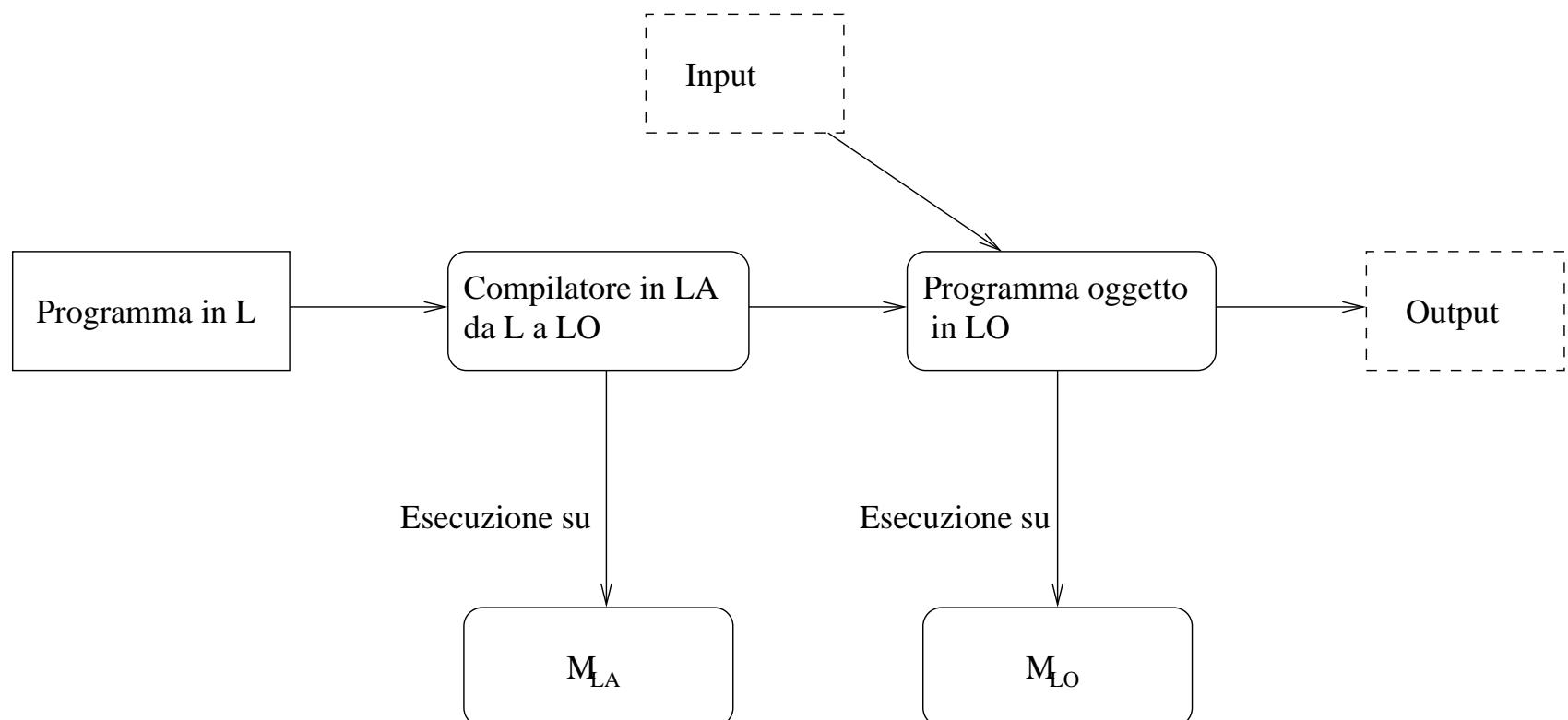
Compilazione ed ...

Compilatore

Proprietà dei ...

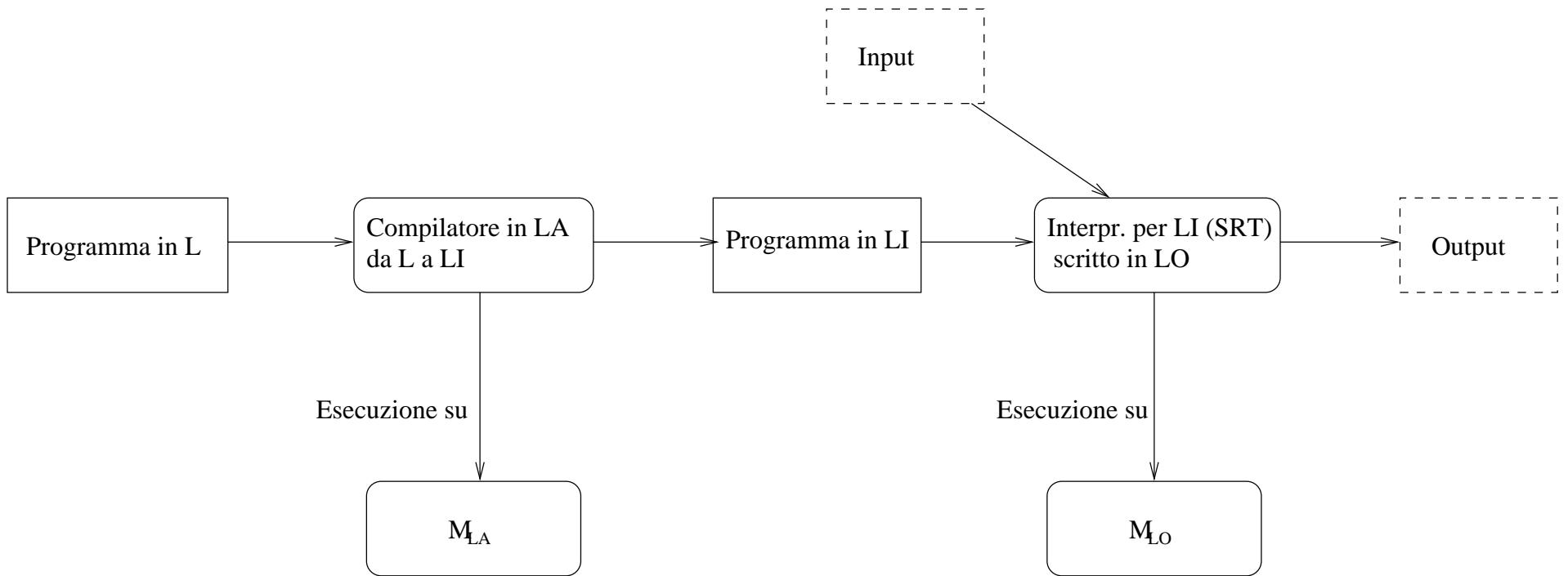
Criteri di scelta ...

Paradigmi  
computazionali





# Compilazione per macchina intermedia





# Supporto a Run Time (SRT)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

**SRT**

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

## ■ Funzionalità aggiuntive (rispetto a $M_{LO}$ )

- ◆ Funzioni di basso livello / interfacce col S.O.; ad es. funzioni di I/O



# Supporto a Run Time (SRT)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

**SRT**

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

## ■ Funzionalità aggiuntive (rispetto a $M_{LO}$ )

- ◆ Funzioni di basso livello / interfacce col S.O.; ad es. funzioni di I/O
- ◆ Gestione della memoria; garbage collection; gestione dell'heap; gestione dello stack



# Supporto a Run Time (SRT)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

**SRT**

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

- Funzionalità aggiuntive (rispetto a  $M_{LO}$ )
  - ◆ Funzioni di basso livello / interfacce col S.O.; ad es. funzioni di I/O
  - ◆ Gestione della memoria; garbage collection; gestione dell'heap; gestione dello stack
- Non necessariamente una macchina astratta radicalmente diversa
  - ◆ A volte solo un pacchetto di funzioni o librerie aggiunte automaticamente al codice oggetto



# Compilazione ed esecuzione

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

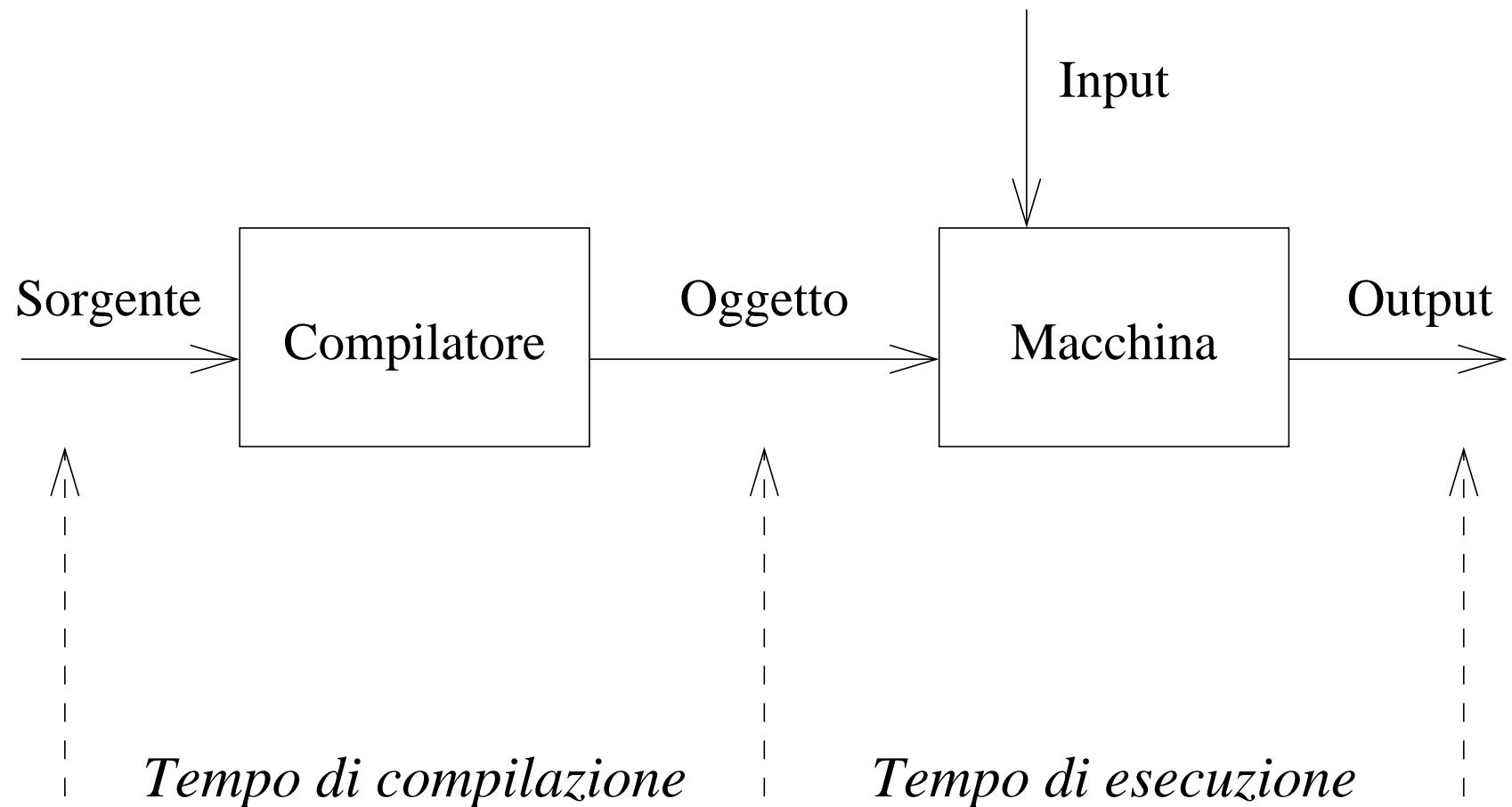
**Compilazione ed ...**

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali





# Compilatore

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

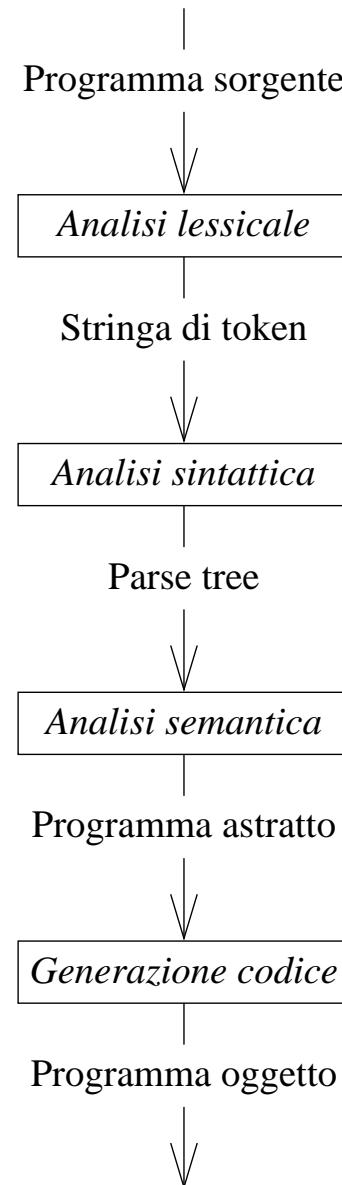
Compilazione ed ...

**Compilatore**

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali





# Proprietà dei linguaggi

## ■ Semplicità – (concisione) VS (leggibilità)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

## ■ Semplicità – (concisione) VS (leggibilità)

- ◆ Semantica: minimo numero di concetti e strutture.



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

## ■ Semplicità – (concisione) VS (leggibilità)

- ◆ Semantica: minimo numero di concetti e strutture.
- ◆ Sintattica: unica rappresentabilità di ogni concetto.



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

- Semplicità – (concisione) VS (leggibilità)
  - ◆ Semantica: minimo numero di concetti e strutture.
  - ◆ Sintattica: unica rappresentabilità di ogni concetto.
- Astrazione – (rappresentare solo attributi essenziali)



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

## ■ Semplicità – (concisione) 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.



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

- Semplicità – (concisione) 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.



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

- Semplicità – (concisione) 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)



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

- Semplicità – (concisione) 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 alle regole del linguaggio)



# Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

Criteri di scelta ...

Paradigmi  
computazionali

- Semplicità – (concisione) 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 alle regole del linguaggio)
- Portabilità



# Criteri di scelta del linguaggio

## ■ Disponibilità dei traduttori

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

**Criteri di scelta ...**

Paradigmi  
computazionali



# Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

**Criteri di scelta ...**

Paradigmi  
computazionali



# Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore
- Esistenza di standard di portabilità

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

**Criteri di scelta ...**

Paradigmi  
computazionali



# Criteri di scelta del linguaggio



Disponibilità dei traduttori



Maggiore conoscenza da parte del programmatore



Esistenza di standard di portabilità



Comodità dell'ambiente di programmazione



Sintassi aderente al problema

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

**Criteri di scelta ...**

Paradigmi  
computazionali



# Criteri di scelta del linguaggio

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore
- Esistenza di standard di portabilità
- Comodità dell'ambiente di programmazione
- Sintassi aderente al problema
- Semantica aderente alla architettura fisica?

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

**Criteri di scelta ...**

Paradigmi  
computazionali



# Criteri di scelta del linguaggio

Introduzione al corso

Linguaggi (di  
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Realizzazione M.A.

Traduttori

Linguaggio →  
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

SRT

Compilazione ed ...

Compilatore

Proprietà dei ...

**Criteri di scelta ...**

Paradigmi  
computazionali

- Disponibilità dei traduttori
- Maggiore conoscenza da parte del programmatore
- Esistenza di standard di portabilità
- Comodità dell'ambiente di programmazione
- Sintassi aderente al problema
- Semantica aderente alla architettura fisica?
  - ◆ con le moderne tecniche di implementazione dei linguaggi non è più un criterio stringente



[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

**Paradigmi  
computazionali**

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Conclusioni

# Paradigmi computazionali



# Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Conclusioni

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



# Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

**Paradigmi**

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Conclusioni

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

**Funzionale:** Il programma e le sue componenti sono *funzioni*.  
Esecuzione come valutazione di funzioni.



# Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Conclusioni

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

**Funzionale:** Il programma e le sue componenti sono *funzioni*.  
Esecuzione come valutazione di funzioni.

**Logico:** Programma come descrizione logica di un problema.  
Esecuzione analoga a processi di dimostrazione di teoremi.



# Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Conclusioni

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

**Funzionale:** Il programma e le sue componenti sono *funzioni*.  
Esecuzione come valutazione di funzioni.

**Logico:** Programma come descrizione logica di un problema.  
Esecuzione analoga a processi di dimostrazione di teoremi.

**Orientato ad oggetti:** Programma costituito da oggetti che scambiano messaggi.



# Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di  
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi  
computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Conclusioni

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

**Funzionale:** Il programma e le sue componenti sono *funzioni*.  
Esecuzione come valutazione di funzioni.

**Logico:** Programma come descrizione logica di un problema.  
Esecuzione analoga a processi di dimostrazione di teoremi.

**Orientato ad oggetti:** Programma costituito da oggetti che scambiano messaggi.

**Parallello:** Programmi che descrivono entità distribuite che sono eseguite contemporaneamente ed in modo asincrono.

Gli ultimi due sono ortogonali rispetto ai primi tre...



# Esempio 1.0 Imperativo vs. Funzionale

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi](#)

**Esempio 1.0  
Imperativo vs.  
Funzionale**

[Esempio 1.1](#)

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Conclusioni](#)

## function definition

```
function factI (n)
 local accumulator = 1
 for i = 1,n do
 accum = accumulator*i
 end
 return accum
end
```

## trace of execution

```
factI(4):
 accumulator = 1
 i = 1 accumulator = 1 * 1
 i = 2 accumulator = 1 * 2
 i = 3 accumulator = 2 * 3
 i = 4 accumulator = 6 * 4
 return 24
```

## function definition

```
function factR (n)
 if n == 1 then
 return 1
 else
 return n*factR(n-1)
 end
end
```

## trace of execution

```
factR(4) =
4 * factR(3) =
 3 * factR(2) =
 2 * factR(1) =
 1
 1
 1
 2
 6
```

24

Notare eliminazione assegnamenti:  $n$  rimpiazza  $i$ , ricorsione invece di cicli



## Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi  
Esempio 1.0  
Imperativo vs.  
Funzionale](#)

**Esempio 1.1**

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Conclusioni](#)



## Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

```
membro(2, [1, 2, 3]) = true;
membro(4, [1, 2, 3]) = false.
```

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi  
Esempio 1.0  
Imperativo vs.  
Funzionale](#)

**Esempio 1.1**

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Conclusioni](#)



## Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

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

`membro(4, [1, 2, 3]) = false.`

A questo scopo, si suppongano già esistenti le funzioni:

- `vuota(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi](#)

[Esempio 1.0](#)

[Imperativo vs.](#)

[Funzionale](#)

**Esempio 1.1**

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Conclusioni](#)



## Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

```
membro(2, [1, 2, 3]) = true;
membro(4, [1, 2, 3]) = false.
```

A questo scopo, si suppongano già esistenti le funzioni:

- `vuota(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.
- `testa(L)`, che restituisce il primo elemento della lista `L`.

Es.: `testa([1, 2, 3]) = 1.`

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi](#)

[Esempio 1.0](#)

[Imperativo vs.](#)

[Funzionale](#)

**Esempio 1.1**

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Conclusioni](#)



## Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `membro(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

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

```
membro(4, [1, 2, 3]) = false.
```

A questo scopo, si suppongano già esistenti le funzioni:

- `vuota(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.
- `testa(L)`, che restituisce il primo elemento della lista `L`.  
Es.: `testa([1, 2, 3]) = 1.`
- `coda(L)`, che restituisce una sottolista ottenuta rimuovendo il primo elemento di `L`.  
Es.: `coda([1, 2, 3]) = [2, 3].`



## Esempio 1.2

La funzione membro nel paradigma imperativo:

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

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale

Esempio 1.1

**Esempio 1.2**

Esempio 2

Esempio 3

Conclusioni



## Esempio 1.2

La funzione membro nel paradigma imperativo:

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

In C (altro linguaggio imperativo):

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

NB: nessuna differenza strutturale, solo dettagli sintattici



## Esempio 2

La funzione membro nel paradigma funzionale (come Lisp):

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

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

**Esempio 2**

Esempio 3

Conclusioni



## Esempio 2

La funzione membro nel paradigma funzionale (come Lisp):

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

Nella versione funzionale pura non ci sono variabili, nè assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione. La sintassi Lisp e Scheme sarebbe un po' particolare:

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

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

**Esempio 2**

Esempio 3

Conclusioni



## Esempio 2

La funzione membro nel paradigma funzionale (come Lisp):

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

Nella versione funzionale pura non ci sono variabili, nè assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione. La sintassi Lisp e Scheme sarebbe un po' particolare:

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

Anche il C si potrebbe usare in stile funzionale se evitassimo di usare i costrutti imperativi:

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



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.  
Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

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

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale  
Esempio 1.1  
Esempio 1.2  
Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- `member(2,[1,2,3])` restituisce *yes* (`true`)
- `member(0,[1,2,3])` restituisce *no* (`false`)
- query con variabili: `member(X,[1,2,3])` restituisce
  - ◆ `X=1`

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale  
Esempio 1.1

Esempio 1.2

Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce yes (true)
- member(0,[1,2,3]) restituisce no (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - ◆ X=1 ; X=2

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale  
Esempio 1.1

Esempio 1.2

Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- `member(2,[1,2,3])` restituisce *yes* (`true`)
- `member(0,[1,2,3])` restituisce *no* (`false`)
- query con variabili: `member(X,[1,2,3])` restituisce
  - ◆ `X=1 ; X=2 ; X=3`

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale  
Esempio 1.1

Esempio 1.2

Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce yes (true)
- member(0,[1,2,3]) restituisce no (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - ◆ X=1; X=2; X=3; no (si comporta come un generatore)

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale  
Esempio 1.1

Esempio 1.2

Esempio 2  
**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce yes (true)
- member(0,[1,2,3]) restituisce no (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - ◆ X=1; X=2; X=3; no (si comporta come un generatore)
- risposte con variabili: member(1,L) restituisce
  - ◆ L=[1 | L<sub>0</sub>]

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale  
Esempio 1.1

Esempio 1.2

Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce yes (true)
- member(0,[1,2,3]) restituisce no (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - ◆ X=1; X=2; X=3; no (si comporta come un generatore)
- risposte con variabili: member(1,L) restituisce
  - ◆ L=[1 | L<sub>0</sub>] ; L=[Y<sub>0</sub>,1 | L<sub>1</sub>]

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi  
Esempio 1.0  
Imperativo vs.](#)

[Funzionale  
Esempio 1.1](#)

[Esempio 1.2](#)

[Esempio 2](#)

**Esempio 3**

[Conclusioni](#)



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- member(2,[1,2,3]) restituisce yes (true)
- member(0,[1,2,3]) restituisce no (false)
- query con variabili: member(X,[1,2,3]) restituisce
  - ◆ X=1; X=2; X=3; no (si comporta come un generatore)
- risposte con variabili: member(1,L) restituisce
  - ◆ L=[1|L<sub>0</sub>]; L=[Y<sub>0</sub>,1|L<sub>1</sub>]; L=[Y<sub>0</sub>,Y<sub>1</sub>,1|L<sub>2</sub>] ...

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.

Funzionale  
Esempio 1.1

Esempio 1.2

Esempio 2

**Esempio 3**

Conclusioni



## Esempio 3

La funzione membro nel paradigma logico (prolog):

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

Nota: i parametri formali di *member* possono essere *pattern*. I programmi consistono di definizioni di *predicati*. Esecuzione:

- `member(2,[1,2,3])` restituisce *yes* (`true`)
- `member(0,[1,2,3])` restituisce *no* (`false`)
- query con variabili: `member(X,[1,2,3])` restituisce
  - ◆ `X=1 ; X=2 ; X=3 ; no` (si comporta come un generatore)
- risposte con variabili: `member(1,L)` restituisce
  - ◆ `L=[1|L0] ; L=[Y0,1|L1] ; L=[Y0,Y1,1|L2] ...`
- *Invertibilità*: nessuna distinzione tra input e output. Un solo predicato (programma), molte funzioni.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi  
Esempio 1.0  
Imperativo vs.](#)

[Funzionale  
Esempio 1.1](#)

[Esempio 1.2](#)

[Esempio 2](#)

**Esempio 3**

[Conclusioni](#)



# Conclusioni

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi  
Esempio 1.0  
Imperativo vs.  
Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Conclusioni

- Il paradigma di appartenenza può influenzare *radicalmente* il modo in cui si risolve il problema
  - ◆ mentre in linguaggi dello stesso paradigma lo stesso problema ha soluzioni strutturalmente identiche
  - ◆ imparato a risolvere un problema in un linguaggio, lo si sa risolvere in tutti i linguaggi dello stesso paradigma
- Il paradigma non è l'unico aspetto determinante. Altri esempi di aspetti importanti:
  - ◆ Il sistema di tipi supportato
  - ◆ Eventuale supporto alle eccezioni
  - ◆ Modello di concorrenza e sincronizzazione
  - ◆ ...

---

# Linguaggi di Programmazione I – Lezione 2

Prof. Marcello Sette  
<mailto://marcello.sette@gmail.com>  
<http://sette.dnsalias.org>

11 marzo 2010



# Panoramica della lezione

**Modello imperativo**

**Data Object e legami**

**Legami di tipo**

**Blocchi di istruzioni**

**Legami di nome**

**Legami di locazione**

**Bibliografia**



Modello imperativo

Memoria

Assegnazioni

Modello imperativo

Nomi

Ambiente

Esempi di ambiente

Esempio:

assegnazione

Data Object e  
legami

---

Legami di tipo

---

Blocchi di istruzioni

---

Legami di nome

---

Legami di locazione

---

Bibliografia

---

# Modello imperativo



# Memoria

Modello imperativo

**Memoria**

Assegnazioni

Modello imperativo

Nomi

Ambiente

Esempi di ambiente

Esempio:

assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

■ Consiste in un insieme di “contenitori di dati” ...

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



# Memoria

[Modello imperativo](#)

**Memoria**

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:  
assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

■ Consiste in un insieme di “contenitori di dati” ...

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

■ ... associati ai valori in essi contenuti

- ◆ I valori delle variabili



# Memoria

Modello imperativo

Memoria

Assegnazioni

Modello imperativo

Nomi

Ambiente

Esempi di ambiente

Esempio:  
assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

■ Consiste in un insieme di “contenitori di dati” ...

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

■ ... associati ai valori in essi contenuti

- ◆ I valori delle variabili

■ Dunque (concettualmente) la memoria è

- ◆ Una funzione da uno spazio di locazioni ad uno spazio di valori
- ◆  $\text{mem(loc)} = \text{"valore contenuto in loc"}$



# Assegnazioni

## ■ Definizione grammaticale (sintassi):

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

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.



# Assegnazioni

## ■ Definizione grammaticale (sintassi):

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

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

Esempio in Pascal:

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



# Assegnazioni

## ■ Definizione grammaticale (sintassi):

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

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

Esempio in Pascal:

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

## ■ Esecuzione (significato, semantica):

Il valore di <expression> va memorizzato nell'indirizzo rappresentato da <name>.



# Assegnazioni

## ■ Definizione grammaticale (sintassi):

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

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

Esempio in Pascal:

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

## ■ Esecuzione (significato, semantica):

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 prescrizioni del codice associato al suo nome.



# Modello imperativo

[Modello imperativo](#)

Memoria

Assegnazioni

**Modello imperativo**

Nomi

Ambiente

Esempi di ambiente

Esempio:

assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.



# Modello imperativo

[Modello imperativo](#)

Memoria

Assegnazioni

**Modello imperativo**

Nomi

Ambiente

Esempi di ambiente

Esempio:

assegnazione

Data Object e  
legami

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.
- I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore.



# Modello imperativo

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:  
assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.
- I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore.
- Ogni unità di esecuzione consiste di 4 passi:
  1. ottenere indirizzi delle locazioni di operandi e risultato;



# Modello imperativo

[Modello imperativo](#)

Memoria

Assegnazioni

**Modello imperativo**

Nomi

Ambiente

Esempi di ambiente

Esempio:  
assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.
- I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore.
- Ogni unità di esecuzione consiste di 4 passi:
  1. ottenere indirizzi delle locazioni di operandi e risultato;
  2. ottenere dati di operandi da locazioni di operandi;



# Modello imperativo

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:  
assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.
- I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore.
- Ogni unità di esecuzione consiste di 4 passi:
  1. ottenere indirizzi delle locazioni di operandi e risultato;
  2. ottenere dati di operandi da locazioni di operandi;
  3. valutare risultato;



# Modello imperativo

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

**Modello imperativo**

[Nomi](#)

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:  
assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.
- I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore.
- Ogni unità di esecuzione consiste di 4 passi:
  1. ottenere indirizzi delle locazioni di operandi e risultato;
  2. ottenere dati di operandi da locazioni di operandi;
  3. valutare risultato;
  4. memorizzare risultato in locazione risultato.



# Modello imperativo

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:  
assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Simula le azioni dell'elaboratore a livello di linguaggio macchina.
- I programmi sono descrizioni di sequenze di modifiche della "memoria" del calcolatore.
- Ogni unità di esecuzione consiste di 4 passi:
  1. ottenere indirizzi delle locazioni di operandi e risultato;
  2. ottenere dati di operandi da locazioni di operandi;
  3. valutare risultato;
  4. memorizzare risultato in locazione risultato.
- Si caratterizza per l'uso dei nomi come astrazione di indirizzi di locazioni di memoria.



# Nomi (di variabili o di parametri di procedure)

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

**Nomi**

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:](#)

[assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

■ Le variabili di un programma costituiscono una memoria?



# Nomi (di variabili o di parametri di procedure)

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

**Nomi**

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:](#)

[assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Le variabili di un programma costituiscono una memoria?
- E i parametri formali delle funzioni/procedure?



# Nomi (di variabili o di parametri di procedure)

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

**Nomi**

[Ambiente](#)

[Esempi di ambiente](#)

[Esempio:  
assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Le variabili di un programma costituiscono una memoria?
- E i parametri formali delle funzioni/procedure? Non proprio (sia concettualmente che per come sono realizzati)



# Ambiente (di esecuzione)

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

**Ambiente**

[Esempi di ambiente](#)

[Esempio:](#)

[assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

■ Insieme di nomi di variabili e parametri ...

◆ Non indirizzi di memoria, piuttosto identificatori



# Ambiente (di esecuzione)

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

**Ambiente**

[Esempi di ambiente](#)

Esempio:  
assegnazione

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

■ Insieme di nomi di variabili e parametri ...

◆ Non indirizzi di memoria, piuttosto identificatori

■ ... associati a qualcosa da cui si può risalire al valore della variabile o del parametro.



# Ambiente (di esecuzione)

Modello imperativo

Memoria

Assegnazioni

Modello imperativo

Nomi

**Ambiente**

Esempi di ambiente

Esempio:  
assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Insieme di nomi di variabili e parametri ...

- ◆ Non indirizzi di memoria, piuttosto identificatori

- ... associati a qualcosa da cui si può risalire al valore della variabile o del parametro.

- Concettualmente l'ambiente è: una funzione da un insieme di identificatori (i nomi) a un insieme di ...?

$\text{env(id)} = ???$



# Ambiente (di esecuzione)

Modello imperativo

Memoria

Assegnazioni

Modello imperativo

Nomi

**Ambiente**

Esempi di ambiente

Esempio:  
assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Insieme di nomi di variabili e parametri ...

- ◆ Non indirizzi di memoria, piuttosto identificatori

- ... associati a qualcosa da cui si può risalire al valore della variabile o del parametro.

- Concettualmente l'ambiente è: una funzione da un insieme di identificatori (i nomi) a un insieme di ...?

$\text{env(id)} = ???$

- Il codominio della funzione dipende dal paradigma computazionale del linguaggio.



# Esempi di ambiente

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

**[Esempi di ambiente](#)**

[Esempio:](#)

[assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Nel paradigma imperativo, la funzione env associa gli identificatori a locazioni di memoria, le quali, a loro volta, sono associate (funzione mem) al contenuto di memoria



# Esempi di ambiente

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

**Esempi di ambiente**

[Esempio:](#)

[assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Nel paradigma imperativo, la funzione env 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  $\text{mem}(\text{env}(x))$



# Esempi di ambiente

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

**Esempi di ambiente**

[Esempio:](#)

[assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Nel paradigma imperativo, la funzione env 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  $\text{mem}(\text{env}(x))$

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



# Esempi di ambiente

[Modello imperativo](#)

[Memoria](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Nomi](#)

[Ambiente](#)

**Esempi di ambiente**

[Esempio:  
assegnazione](#)

[Data Object e  
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

- Nel paradigma imperativo, la funzione env 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  $\text{mem}(\text{env}(x))$

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

- Attenzione:  $\text{env}(x)$  è immutabile finchè  $x$  esiste...  
nel paradigma imperativo la funzione env identifica una associazione *immutabile* (la locazione di memoria associata a un nome non cambia);



# Esempi di ambiente

Modello imperativo

Memoria

Assegnazioni

Modello imperativo

Nomi

Ambiente

**Esempi di ambiente**

Esempio:

assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Nel paradigma imperativo, la funzione env 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  $\text{mem}(\text{env}(x))$

- Nel paradigma funzionale, non esiste la funzione mem e la funzione env associa direttamente gli identificatori al contenuto della memoria.
- Attenzione:  $\text{env}(x)$  è immutabile finchè  $x$  esiste...

nel paradigma imperativo la funzione env identifica una associazione *immutabile* (la locazione di memoria associata a un nome non cambia); anche nel paradigma funzionale puro l'associazione identificatore-valore non cambia



# Esempio: assegnazione

Modello imperativo

Memoria

Assegnazioni

Modello imperativo

Nomi

Ambiente

Esempi di ambiente

Esempio:  
assegnazione

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

In una assegnazione:

```
x := x + 1;
```

la x di sinistra indica la locazione associata al nome (cioè  $\text{env}(x)$ )

la x di destra indica il valore della variabile (cioè  $\text{mem}(\text{env}(x))$ )



Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

## Data Object e legami



# Data Object

[Modello imperativo](#)

[Data Object e legami](#)

**Data Object**

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

■ Un *data object* è la quadrupla  $(L, N, V, T)$ , ove:

- ◆  $L$ : locazione.
- ◆  $N$ : nome.
- ◆  $V$ : valore.
- ◆  $T$ : tipo.



# Data Object

[Modello imperativo](#)

[Data Object e legami](#)

**Data Object**

[Legami](#)

[Modifiche di legami](#)

[Esempio 1](#)

[Esempio 2](#)

[Esempio 3](#)

[Il puntatore \(1\)](#)

[Il puntatore \(2\)](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

■ Un *data object* è la quadrupla  $(L, N, V, T)$ , ove:

- ◆  $L$ : locazione.
- ◆  $N$ : nome.
- ◆  $V$ : valore.
- ◆  $T$ : tipo.

■ Un *legame* è la determinazione di una delle componenti.



# Legami

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Legami di tipo

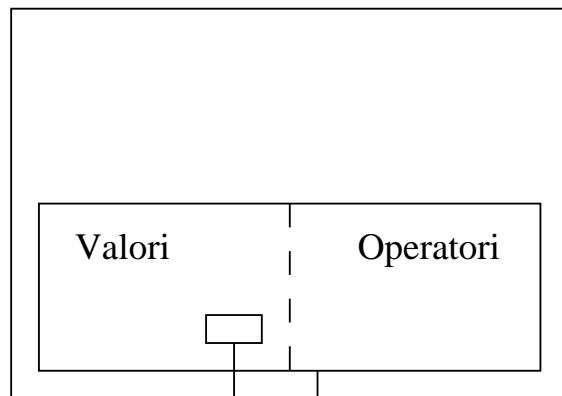
Blocchi di istruzioni

Legami di nome

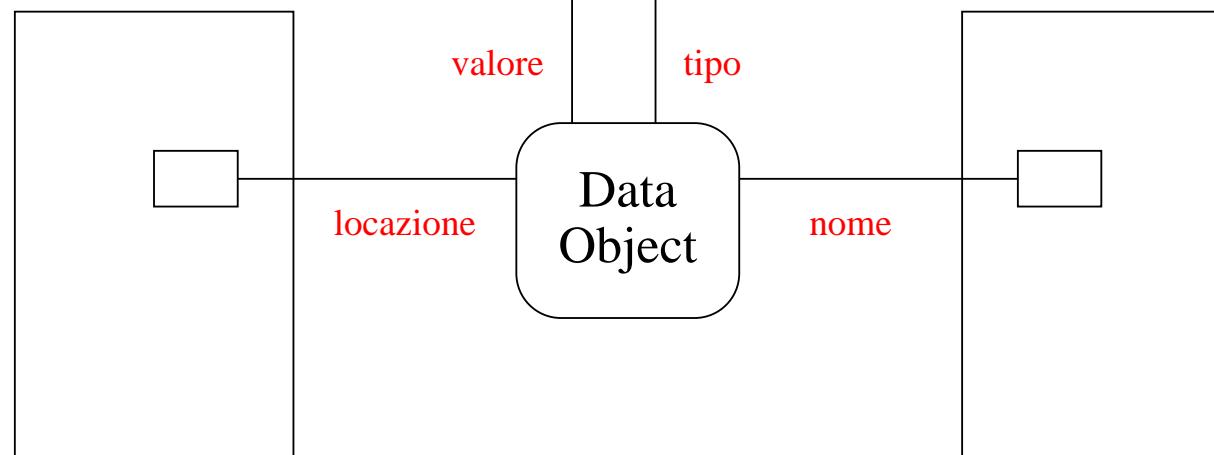
Legami di locazione

Bibliografia

## Spazio di tipi



## Spazio di memoria





# Modifiche di legami

[Modello imperativo](#)

[Data Object e  
legami](#)

[Data Object](#)

[Legami](#)

[\*\*Modifiche di legami\*\*](#)

[Esempio 1](#)

[Esempio 2](#)

[Esempio 3](#)

[Il puntatore \(1\)](#)

[Il puntatore \(2\)](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).



# Modifiche di legami

[Modello imperativo](#)

[Data Object e legami](#)

[Data Object](#)

[Legami](#)

[\*\*Modifiche di legami\*\*](#)

[Esempio 1](#)

[Esempio 2](#)

[Esempio 3](#)

[Il puntatore \(1\)](#)

[Il puntatore \(2\)](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).



# Modifiche di legami

[Modello imperativo](#)

[Data Object e legami](#)

[Data Object](#)

[Legami](#)

[\*\*Modifiche di legami\*\*](#)

[Esempio 1](#)

[Esempio 2](#)

[Esempio 3](#)

[Il puntatore \(1\)](#)

[Il puntatore \(2\)](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).



# Modifiche di legami

[Modello imperativo](#)

[Data Object e legami](#)

[Data Object](#)

[Legami](#)

[\*\*Modifiche di legami\*\*](#)

[Esempio 1](#)

[Esempio 2](#)

[Esempio 3](#)

[Il puntatore \(1\)](#)

[Il puntatore \(2\)](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).

- il *location binding* avviene durante il caricamento in memoria, oppure a run-time (si veda la gestione dei blocchi più avanti);



# Modifiche di legami

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).

- il *location binding* avviene durante il caricamento in memoria, oppure a run-time (si veda la gestione dei blocchi più avanti);
- il *name binding* avviene durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione;



# Modifiche di legami

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
  2. Durante il caricamento in memoria (load time).
  3. Durante l'esecuzione (run time).
- 
- il *location binding* avviene durante il caricamento in memoria, oppure a run-time (si veda la gestione dei blocchi più avanti);
  - il *name binding* avviene durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione;
  - il *type binding* avviene (di solito, si veda dopo) durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione di tipo; un tipo è definito dal sottospazio di valori (e dai relativi operatori) che un *data object* può assumere.



# Esempio 1

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

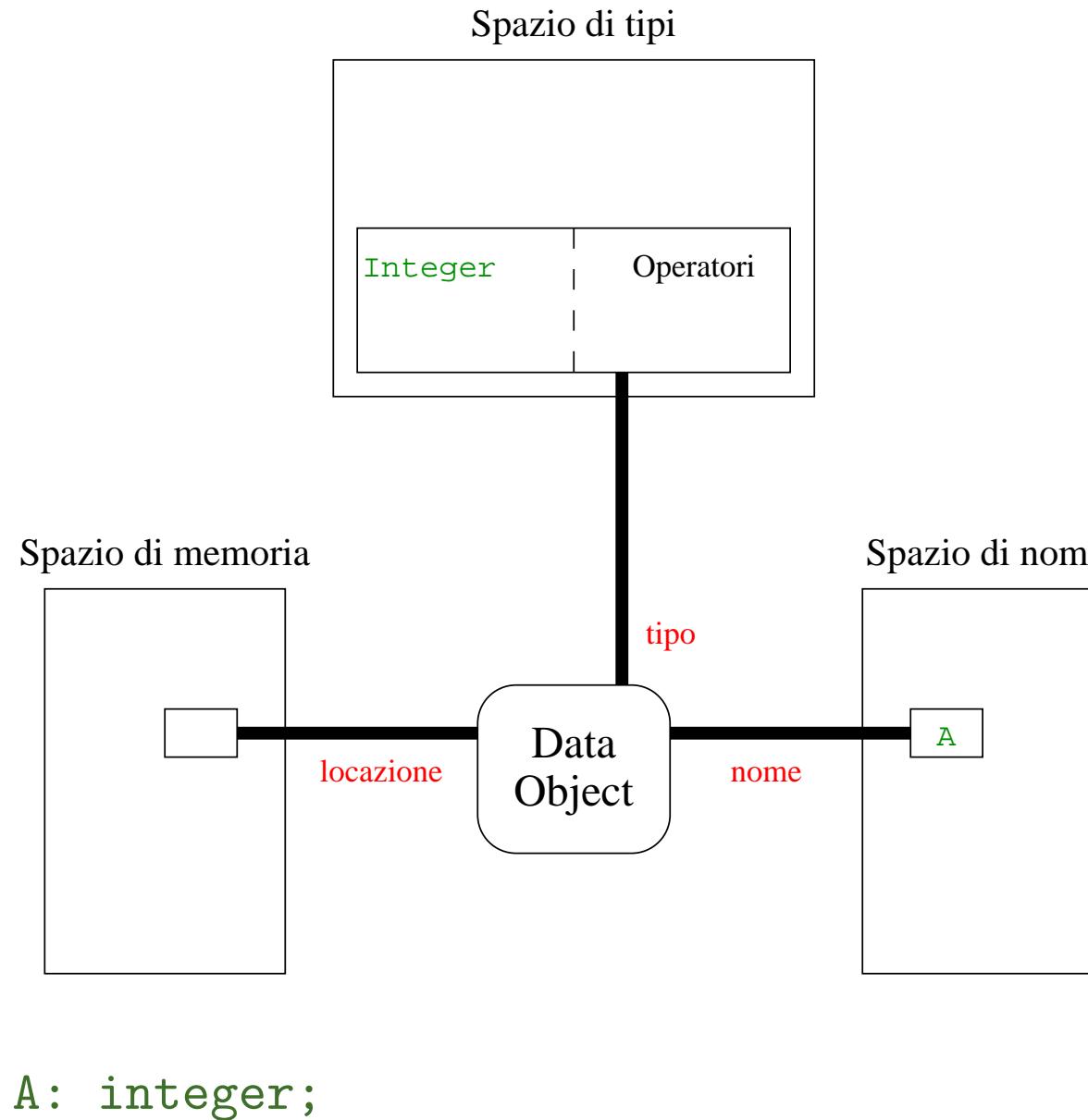
Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia





## Esempio 2

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

**Esempio 2**

Esempio 3

Il puntatore (1)

Il puntatore (2)

Legami di tipo

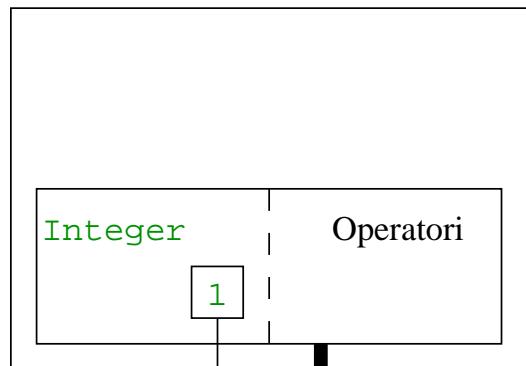
Blocchi di istruzioni

Legami di nome

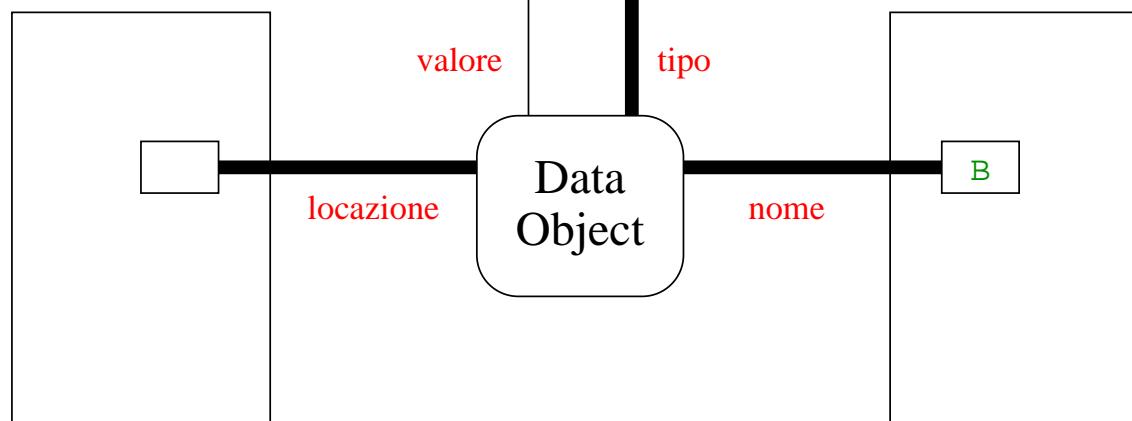
Legami di locazione

Bibliografia

Spazio di tipi



Spazio di memoria



B: integer:= 1;



## Esempio 3

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

**Esempio 3**

Il puntatore (1)

Il puntatore (2)

Legami di tipo

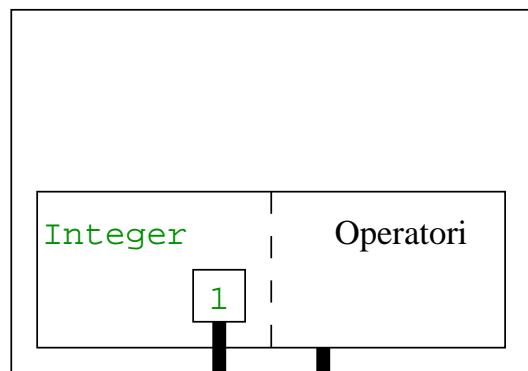
Blocchi di istruzioni

Legami di nome

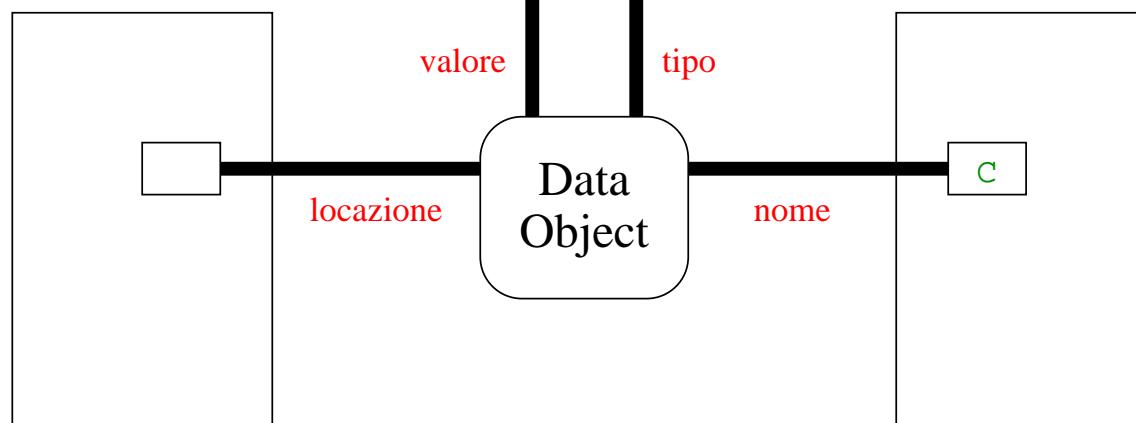
Legami di locazione

Bibliografia

Spazio di tipi



Spazio di memoria



C: constant integer:= 1;



# Il puntatore (1)

Modello imperativo

Data Object e legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

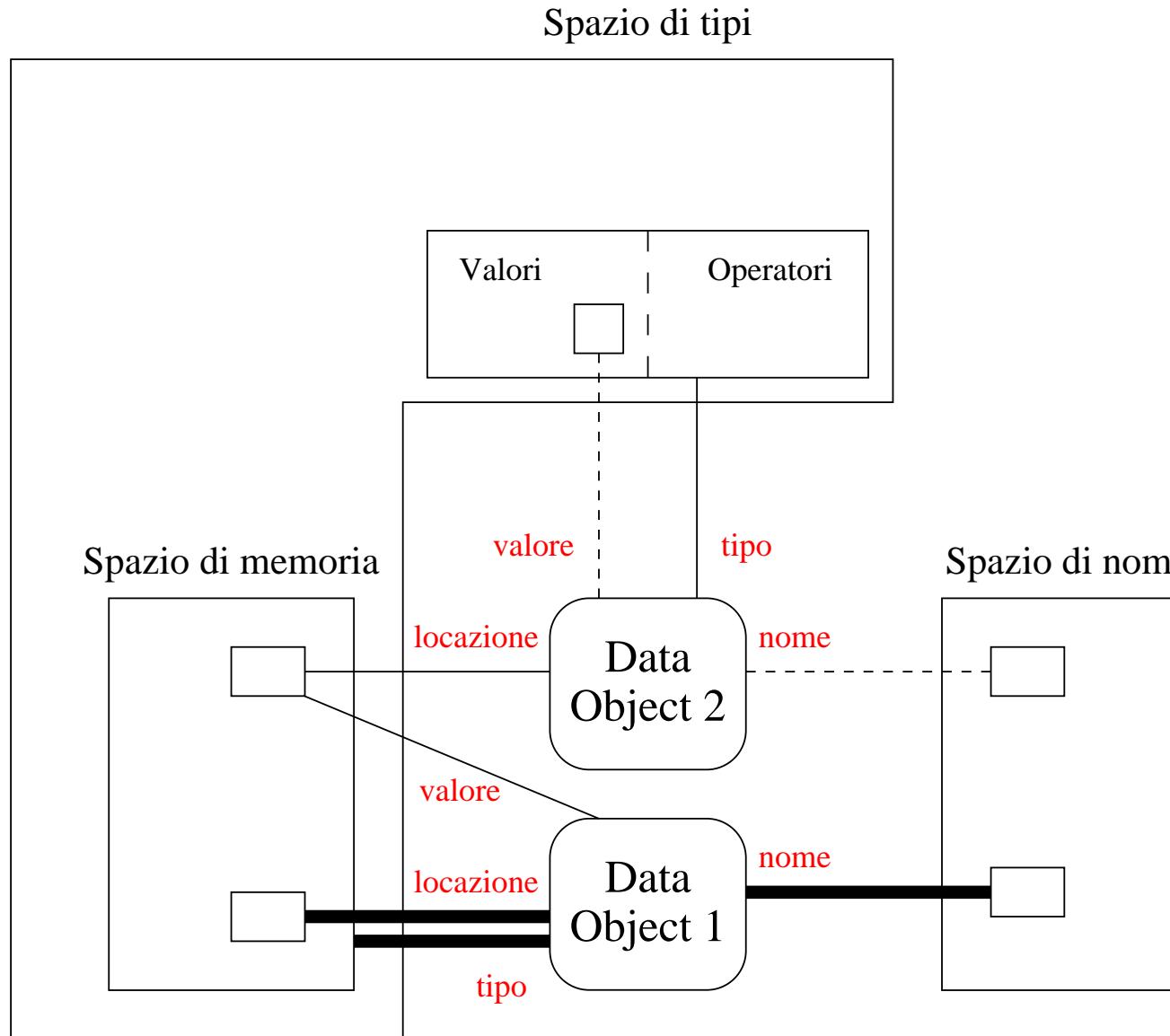
Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia





# Il puntatore (2)

■ 2 *data object* coinvolti.

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia



## Il puntatore (2)

- 2 *data object* coinvolti.
- Il secondo può non avere un legame di nome o di valore.

[Modello imperativo](#)

[Data Object e  
legami](#)

[Data Object](#)

[Legami](#)

[Modifiche di legami](#)

[Esempio 1](#)

[Esempio 2](#)

[Esempio 3](#)

[Il puntatore \(1\)](#)

[\*\*Il puntatore \(2\)\*\*](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)



## Il puntatore (2)

Modello imperativo

Data Object e  
legami

Data Object

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

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



## Il puntatore (2)

- 2 *data object* coinvolti.
- Il secondo può non avere un legame di nome o di valore.
- La deallocazione è necessaria, perché 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 (*garbage collector*).

[Modello imperativo](#)

[Data Object e legami](#)

[Data Object](#)

[Legami](#)

[Modifiche di legami](#)

[Esempio 1](#)

[Esempio 2](#)

[Esempio 3](#)

[Il puntatore \(1\)](#)

[\*\*Il puntatore \(2\)\*\*](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)



[Modello imperativo](#)

[Data Object e legami](#)

[\*\*Legami di tipo\*\*](#)

[Legame di tipo](#)

[Type checking \(1\)](#)

[Type checking \(2\)](#)

[Linguaggio perfetto \(1\)](#)

[Linguaggio perfetto \(2\)](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

## Legami di tipo



## Legame di tipo

- Per definizione è correlato al legame di valore.



## Legame di tipo

- Per definizione è correlato al legame di valore.
- Sia quando si instaura, sia quando viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di valore.



## Legame di tipo

- Per definizione è correlato al legame di valore.
- Sia quando si instaura, sia quando viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di valore.
- Un linguaggio è *dinamicamente tipizzato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.



## Legame di tipo

- Per definizione è correlato al legame di valore.
- Sia quando si instaura, sia quando viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di valore.
- Un linguaggio è *dinamicamente tipizzato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.

Esempio: nei linguaggi di scripting

```
x=1; ... x= "abc";
```

Inizialmente il tipo è numerico, poi è stringa (il legame di tipo cambia in seguito ad un cambio del legame di valore).



## Legame di tipo

- Per definizione è correlato al legame di valore.
- Sia quando si instaura, sia quando viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di valore.
- Un linguaggio è *dinamicamente tipizzato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.

Esempio: nei linguaggi di scripting

```
x=1; ... x= "abc";
```

Inizialmente il tipo è numerico, poi è stringa (il legame di tipo cambia in seguito ad un cambio del legame di valore).

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



# Type checking (1)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.



# Type checking (1)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.



# Type checking (1)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre: il più possibile durante la compilazione e, negli altri casi, durante l'esecuzione.



# Type checking (1)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre: il più possibile durante la compilazione e, negli altri casi, durante l'esecuzione.
  - ◆ Un linguaggio fortemente tipizzato è anche staticamente tipizzato.



# Type checking (1)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre: il più possibile durante la compilazione e, negli altri casi, durante l'esecuzione.
  - ◆ Un linguaggio fortemente tipizzato è anche staticamente tipizzato.
  - ◆ Java è fortemente tipizzato (vedremo poi).



# Type checking (1)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre: il più possibile durante la compilazione e, negli altri casi, durante l'esecuzione.
  - ◆ Un linguaggio fortemente tipizzato è anche staticamente tipizzato.
  - ◆ Java è fortemente tipizzato (vedremo poi).
  - ◆ Pascal è quasi fortemente tipizzato (una sola eccezione di assenza di controllo: i record con varianti).



# Type checking (1)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre: il più possibile durante la compilazione e, negli altri casi, durante l'esecuzione.
  - ◆ Un linguaggio fortemente tipizzato è anche staticamente tipizzato.
  - ◆ Java è fortemente tipizzato (vedremo poi).
  - ◆ Pascal è quasi fortemente tipizzato (una sola eccezione di assenza di controllo: i record con varianti).
  - ◆ Linguaggi didatticamente rilevanti.



## Type checking (2)

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto (1)

Linguaggio perfetto (2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia



## Type checking (2)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.
  - ◆ C è debolmente tipizzato:  
in esso esistono le operazioni di *casting*, che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo (anche un tipo diverso da quello a cui il valore è stato precedentemente associato);



## Type checking (2)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.

- ◆ C è debolmente tipizzato:
  - in esso esistono le operazioni di *casting*, che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo (anche un tipo diverso da quello a cui il valore è stato precedentemente associato);
  - esistono puntatori a void, che godono, in esecuzione, di conversione di tipo implicita verso qualunque altro tipo puntatore;



## Type checking (2)

Modello imperativo

Data Object e  
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Linguaggio perfetto  
(1)

Linguaggio perfetto  
(2)

Blocchi di istruzioni

Legami di nome

Legami di locazione

Bibliografia

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.
  - ◆ C è debolmente tipizzato:
    - in esso esistono le operazioni di *casting*, che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo (anche un tipo diverso da quello a cui il valore è stato precedentemente associato);
    - esistono puntatori a void, che godono, in esecuzione, di conversione di tipo implicita verso qualunque altro tipo puntatore;
    - esistono le *unioni*, che consentono di interpretare una collezione di dati correlati secondo diverse attribuzioni di tipo indipendenti.



## Il linguaggio perfetto (1)

Sarebbe bello se esistesse un linguaggio Turing completo, in cui il controllo di consistenza di tipo avvenisse completamente durante la compilazione e *in cui il compilatore non generasse più errori del necessario* – Linguaggio Perfetto (LP).

- LP, se esistesse, sarebbe staticamente e fortemente tipizzato (il più “forte” di tutti i fortemente tipizzati).

[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Legame di tipo](#)

[Type checking \(1\)](#)

[Type checking \(2\)](#)

[\*\*Linguaggio perfetto \(1\)\*\*](#)

[\*\*Linguaggio perfetto \(2\)\*\*](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)



## Il linguaggio perfetto (1)

[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Legame di tipo](#)

[Type checking \(1\)](#)

[Type checking \(2\)](#)

[Linguaggio perfetto \(1\)](#)

[Linguaggio perfetto \(2\)](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

Sarebbe bello se esistesse un linguaggio Turing completo, in cui il controllo di consistenza di tipo avvenisse completamente durante la compilazione e *in cui il compilatore non generasse più errori del necessario* – Linguaggio Perfetto (LP).

- LP, se esistesse, sarebbe staticamente e fortemente tipizzato (il più “forte” di tutti i fortemente tipizzati).
- LP non può esistere.



## Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

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

dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P, quindi LP non sarebbe Turing completo, contro l'ipotesi.



## Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

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

dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P, quindi LP non sarebbe Turing completo, contro l'ipotesi.

- I compilatori, in situazioni come la precedente, presumono la terminazione di P e segnalano un errore nell'ultima linea.



## Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

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

dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P, quindi LP non sarebbe Turing completo, contro l'ipotesi.

- I compilatori, in situazioni come la precedente, presumono la terminazione di P e segnalano un errore nell'ultima linea.

Per esempio, questo programma Pascal non compila:

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



## Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

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

dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P, quindi LP non sarebbe Turing completo, contro l'ipotesi.

- I compilatori, in situazioni come la precedente, presumono la terminazione di P e segnalano un errore nell'ultima linea.

Per esempio, questo programma Pascal non compila:

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

Ma un tale programma verrebbe correttamente eseguito: il compilatore ha segnalato un errore di troppo.



Modello imperativo

Data Object e  
legami

Legami di tipo

**Blocchi di istruzioni**

Necessità

Definizioni

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia

## Blocchi di istruzioni



## Necessità

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo;

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

Definizioni

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia



## Necessità

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

Definizioni

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo;
- ambito di una procedura;



## Necessità

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

Definizioni

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo;
- ambito di una procedura;
- unità di compilazione separata;



## Necessità

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

Definizioni

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia

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.



## Definizioni

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

**Definizioni**

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia

Blocchi che definiscono l'ambito di validità di un nome contengono due parti:

- una sezione di dichiarazione del nome;



## Definizioni

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

Definizioni

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia

Blocchi che definiscono l'ambito di validità di un nome contengono due parti:

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



## Definizioni

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

Definizioni

Ambito di ...

Legami di nome

Legami di locazione

Bibliografia

Blocchi che definiscono l'ambito di validità di un nome contengono due parti:

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

Definiamo un piccolo pseudo-linguaggio per rappresentare un blocco:

```
...
BLOCK A;
 DECLARE I;
BEGIN A
 ...
 {I DEL BLOCCO A}
END A;
...
```



## Ambito di validità di legami

Essenzialmente due tipi di ambito di validità (scoping):

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità

Definizioni

Ambito di . . .

Legami di nome

Legami di locazione

Bibliografia



## Ambito di validità di legami

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità  
Definizioni

**Ambito di . . .**

Legami di nome

Legami di locazione

Bibliografia

Essenzialmente due tipi di ambito di validità (scoping):

**In ambito statico** o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (*legami non locali*) e, di solito, possono aggiungere legami locali o sovrapporne di nuovi.



# Ambito di validità di legami

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità  
Definizioni

**Ambito di . . .**

Legami di nome

Legami di locazione

Bibliografia

Essenzialmente due tipi di ambito di validità (scoping):

**In ambito statico** o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (*legami non locali*) e, di solito, possono aggiungere legami locali o sovrapporli a nuovi.

**In ambito dinamico**

Concetto qui esaminato solo in relazione ai blocchi annidati, ma che assume il proprio senso maggiore quando vi sono procedure chiamanti e chiamate. In questo caso la procedura chiamata vede e usa i legami visti e usati dalla procedura chiamante.



# Ambito di validità di legami

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Necessità  
Definizioni

**Ambito di . . .**

Legami di nome

Legami di locazione

Bibliografia

Essenzialmente due tipi di ambito di validità (scoping):

**In ambito statico** o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (*legami non locali*) e, di solito, possono aggiungere legami locali o sovrapporne di nuovi.

**In ambito dinamico**

Concetto qui esaminato solo in relazione ai blocchi annidati, ma che assume il proprio senso maggiore quando vi sono procedure chiamanti e chiamate. In questo caso la procedura chiamata vede e usa i legami visti e usati dalla procedura chiamante.

Esamineremo tutti i dettagli nella prossima lezione.



[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[\*\*Legami di nome\*\*](#)

Static scoping

Mascheramento

[Legami di locazione](#)

[Bibliografia](#)

## Legami di nome



# Static scoping

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

**Static scoping**

Mascheramento

Legami di locazione

Bibliografia

```
PROGRAM P;
 DECLARE X;
BEGIN P
 ...
 {X da P}
```



# Static scoping

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

**Static scoping**

Mascheramento

Legami di locazione

Bibliografia

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



# Static scoping

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

**Static scoping**

Mascheramento

Legami di locazione

Bibliografia

```
PROGRAM P;
 DECLARE X;
BEGIN P
 ...
 BLOCK A;
 DECLARE Y;
 BEGIN A
 ...
 BLOCK B;
 DECLARE Z;
 BEGIN B
 ...

```

$\{X \text{ da } P\}$

$\{X \text{ da } P, Y \text{ da } A\}$

$\{X \text{ da } P, Y \text{ da } A, Z \text{ da } B\}$



# Static scoping

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

**Static scoping**

Mascheramento

Legami di locazione

Bibliografia

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



# Static scoping

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

**Static scoping**

Mascheramento

Legami di locazione

Bibliografia

```
PROGRAM P;
 DECLARE X;
BEGIN P
 ...
 BLOCK A;
 DECLARE Y;
 BEGIN A
 ...
 BLOCK B;
 DECLARE Z;
 BEGIN B
 ...
 END B;
 ...
END A;
...
```

{X da P}

{X da P, Y da A}

{X da P, Y da A, Z da B}

{X da P, Y da A}

{X da P}



# Static scoping

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Static scoping

Mascheramento

Legami di locazione

Bibliografia

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



# Static scoping

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Static scoping

Mascheramento

Legami di locazione

Bibliografia

```
PROGRAM P;
 DECLARE X;
BEGIN P
 ...
 BLOCK A;
 DECLARE Y;
 BEGIN A
 ...
 BLOCK B;
 DECLARE Z;
 BEGIN B
 ...
 END B;
 ...
 END A;
 ...
 BLOCK C;
 DECLARE Z;
 START C
 ...
 END C;
 ...
END P;
```

{X da P}

{X da P, Y da A}

{X da P, Y da A, Z da B}

{X da P}

{X da P, Z da C}

{X da P}



# Mascheramento

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Static scoping

**Mascheramento**

Legami di locazione

Bibliografia

```
PROGRAM P;
 DECLARE X, Y;
BEGIN P
 ...
 {X e Y da P}
```



# Mascheramento

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Static scoping

**Mascheramento**

Legami di locazione

Bibliografia

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



# Mascheramento

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Static scoping

**Mascheramento**

Legami di locazione

Bibliografia

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



Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

**Legami di locazione**

Allocaz. statica

Allocaz. dinamica

Stack di attivazione

Esempio

Bibliografia

## Legami di locazione



## Allocazione statica di memoria

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

**Allocaz. statica**

Allocaz. dinamica

Stack di attivazione

Esempio

Bibliografia

Si dice *allocazione statica di memoria* (load-time) 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).



# Allocazione statica di memoria

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

Stack di attivazione

Esempio

Bibliografia

Si dice *allocazione statica di memoria* (load-time) 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).

Se il linguaggio prevede ciò, allora, per esempio:

```
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;
```



# Allocazione dinamica di memoria

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

**Allocaz. dinamica**

Stack di attivazione

Esempio

Bibliografia

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



# Allocazione dinamica di memoria

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

**Allocaz. dinamica**

Stack di attivazione

Esempio

Bibliografia

- Si dice *allocazione dinamica di memoria* (run-time) quando il legame di locazione (e anche di nome) è creato all'inizio dell'esecuzione di un blocco e viene rilasciato a fine esecuzione.
- Essa è realizzata attraverso il *record di attivazione* di un blocco.



# Allocazione dinamica di memoria

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

**Allocaz. dinamica**

Stack di attivazione

Esempio

Bibliografia

- Si dice *allocazione dinamica di memoria* (run-time) quando il legame di locazione (e anche di nome) è creato all'inizio dell'esecuzione di un blocco e viene rilasciato a fine esecuzione.
- Essa è realizzata attraverso il *record di attivazione* di un blocco.
  - ◆ Un record di attivazione contiene tutte le informazioni sull'esecuzione del blocco necessarie per riprendere l'esecuzione dopo che essa è stata sospesa.



# Allocazione dinamica di memoria

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

**Allocaz. dinamica**

Stack di attivazione

Esempio

Bibliografia

- Si dice *allocazione dinamica di memoria* (run-time) quando il legame di locazione (e anche di nome) è creato all'inizio dell'esecuzione di un blocco e viene rilasciato a fine esecuzione.
- Essa è realizzata attraverso il *record di attivazione* di un blocco.
  - ◆ Un record di attivazione contiene tutte le informazioni sull'esecuzione del blocco necessarie per riprendere l'esecuzione dopo che essa è stata sospesa.
  - ◆ Può contenere informazioni complesse (come si vedrà in seguito), 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.



## Stack di attivazione

[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

Allocaz. statica

Allocaz. dinamica

**Stack di attivazione**

Esempio

[Bibliografia](#)

In ogni momento dell'esecuzione lo *stack di attivazione* contiene i record “attivi”:



## Stack di attivazione

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

**Stack di attivazione**

Esempio

Bibliografia

In ogni momento dell'esecuzione lo *stack di attivazione* contiene i record “attivi”:

1. il top dello stack contiene sempre il record del blocco correntemente in esecuzione;



## Stack di attivazione

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

**Stack di attivazione**

Esempio

Bibliografia

In ogni momento dell'esecuzione lo *stack di attivazione* contiene i record “attivi”:

1. il top dello stack contiene sempre il record del blocco correntemente in esecuzione;
2. ogni volta che si entra in un blocco, il record di attivazione del blocco viene posto sullo stack (push);



## Stack di attivazione

In ogni momento dell'esecuzione lo *stack di attivazione* contiene i record “attivi”:

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

[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

Allocaz. statica

Allocaz. dinamica

**Stack di attivazione**

Esempio

[Bibliografia](#)



# Esempio di allocazione dinamica

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

Stack di attivazione

**Esempio**

Bibliografia

## All'ingresso in P

Contenuto dello  
stack di attivazione

|      |   |
|------|---|
| I, J | P |
| null |   |

```
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;
```



# Esempio di allocazione dinamica

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

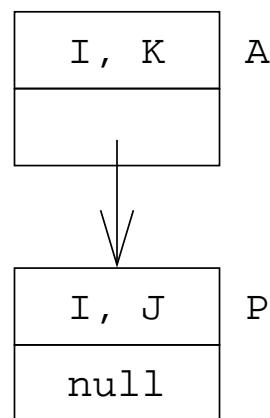
Stack di attivazione

**Esempio**

Bibliografia

## All'ingresso in A

Contenuto dello stack di attivazione



```
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;
```



# Esempio di allocazione dinamica

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

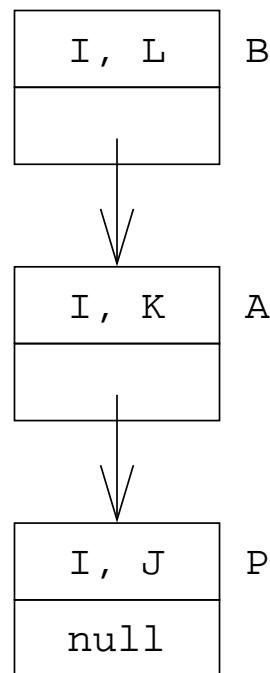
Stack di attivazione

**Esempio**

Bibliografia

## All'ingresso in B

Contenuto dello stack di attivazione



```
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;
```



# Esempio di allocazione dinamica

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

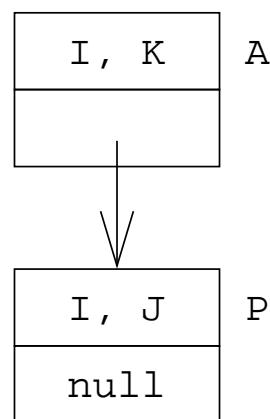
Stack di attivazione

**Esempio**

Bibliografia

## All'uscita da B

Contenuto dello stack di attivazione



```
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;
```



# Esempio di allocazione dinamica

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

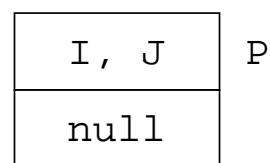
Stack di attivazione

**Esempio**

Bibliografia

## All'uscita da A

Contenuto dello  
stack di attivazione



```
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;
```



# Esempio di allocazione dinamica

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

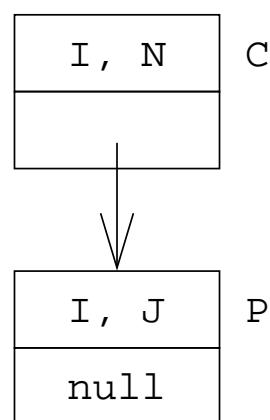
Stack di attivazione

**Esempio**

Bibliografia

## All'ingresso in C

Contenuto dello  
stack di attivazione



```
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;
```



# Esempio di allocazione dinamica

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

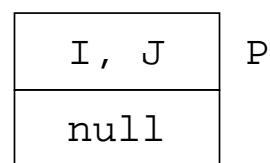
Stack di attivazione

**Esempio**

Bibliografia

## All'uscita da C

Contenuto dello  
stack di attivazione



```
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;
```



# Esempio di allocazione dinamica

Modello imperativo

Data Object e  
legami

Legami di tipo

Blocchi di istruzioni

Legami di nome

Legami di locazione

Allocaz. statica

Allocaz. dinamica

Stack di attivazione

**Esempio**

Bibliografia

## All'uscita da P

Contenuto dello  
stack di attivazione

```
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;
```



# Bibliografia

[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

**Bibliografia**

- Dershem, Jipping. *Programming Languages: Structures and Models*. Cap. 1 e 2; cap. 3, par. 3.1.1 - 3.1.5, 3.1.12, 3.2.



[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di nome](#)

[Legami di locazione](#)

[Bibliografia](#)

**Esercizi**

Esercizio 1

Esercizio 2

## Esercizi



# Esercizio 1

Si considerino i seguenti blocchi annidati. Per ciascun blocco, determinare i legami di ogni variabile. Evidenziare le variabili locali e non locali.

```
PROGRAM P;
 BLOCK B1;
 DECLARE A,B,C;
 BEGIN B1
 BLOCK B2;
 DECLARE C,D;
 BEGIN B2
 BLOCK B3;
 DECLARE B,D,F;
 BEGIN
 ...
 END B3;
 ...
 END B2;
 BLOCK B4;
 DECLARE B,C,D;
 BEGIN B4
 ...
 END B4;
 ...
 END B1;
END P;
```



## Esercizio 2

Tracciare il contenuto dello stack di attivazione durante l'esecuzione del seguente pseudo-programma.

```
PROGRAM P;
 DECLARE X,Y;
BEGIN P
 BLOCK A;
 DECLARE X,Y,Z;
 BEGIN A
 BLOCK B;
 DECLARE Y;
 BEGIN B
 BLOCK C;
 DECLARE X,Y;
 BEGIN C
 ...
 END C;
 BLOCK D;
 DECLARE Z;
 BEGIN D
```

```
 ...
END D;
...
END B;
END A;
BLOCK E;
 DECLARE Z;
BEGIN E
 BLOCK F;
 DECLARE X;
BEGIN F
 ...
END F;
...
END E;
...
END P;
```

# Linguaggi di Programmazione I – Lezione 12

Prof. Marcello Sette  
mailto://marcello.sette@gmail.com  
http://sette.dnsalias.org

20 maggio 2008

|                                |           |
|--------------------------------|-----------|
| <b>Modificatori di accesso</b> | <b>3</b>  |
| Generalità .....               | 4         |
| public .....                   | 5         |
| private (1).....               | 6         |
| private (2).....               | 7         |
| Default .....                  | 8         |
| protected (1).....             | 9         |
| protected (2).....             | 10        |
| Overriding .....               | 11        |
| <b>Altri modificatori</b>      | <b>12</b> |
| final.....                     | 13        |
| abstract .....                 | 14        |
| static .....                   | 15        |
| - Attributi static.....        | 16        |
| - Metodi static.....           | 17        |
| - Blocchi static .....         | 18        |
| native .....                   | 19        |
| transient.....                 | 20        |
| synchronized.....              | 21        |
| volatile .....                 | 22        |
| <b>Sommario</b>                | <b>23</b> |
| Sommario .....                 | 23        |
| <b>Esempio</b>                 | <b>24</b> |
| Forme ricorrenti .....         | 25        |
| Singololetto (1).....          | 26        |
| Singololetto (2).....          | 27        |
| <b>Esercizi</b>                | <b>28</b> |
| Esercizi .....                 | 28        |
| <b>Questionario</b>            | <b>29</b> |
| D 1.....                       | 30        |
| D 2.....                       | 31        |
| D 3.....                       | 32        |

|            |    |
|------------|----|
| D 4 .....  | 33 |
| D 5 .....  | 34 |
| D 6 .....  | 35 |
| D 7 .....  | 36 |
| D 8 .....  | 37 |
| D 9 .....  | 38 |
| D 10 ..... | 39 |

## Modificatori

### Modificatori di accesso

### Altri modificatori

### Sommario

### Esempio

### Esercizi

### Questionario

---

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi.
- Un gruppo di modificatori, detti *di accesso*, specificano a quali classi è permesso usare quella caratteristica.
- Altri modificatori possono essere usati, in combinazione con i precedenti, per qualificare quella caratteristica.
- Per ora descriveremo come i modificatori si applicano alle classi top-level. Le classi *inner* saranno discusse in una successiva lezione.

LP1 – Lezione 12

2 / 39

## Modificatori di accesso

3 / 39

### 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 heap) 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 modificatori di accesso 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`.
- Una caratteristica può avere al più un modificatore di accesso.
- Se non è presente il modificatore, si intende “accesso consentito dallo stesso pacchetto in cui è situata quella caratteristica”.

LP1 – Lezione 12

4 / 39

```
public
```

- È il modificatore più generoso. Accesso consentito a tutte le altre classi.
- Attenzione: l'accesso ad un attributo o un metodo `public` è subordinato all'accesso alla classe che lo contiene.

LP1 – Lezione 12

5 / 39

### private (1)

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

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

 public Complesso(double r, double i) {
 reale=r; immag=i;
 }
 public Complesso add(Complesso c) {
 return new Complesso(reale + c.reale,
 immag + c.immag);
 }
}

public class Cliente {
 void usali() {
 Complesso c1 = new Complesso(1, 2);
 Complesso c2 = new Complesso(3, 4);
 Complesso c3 = c1.add(c2);
 double d = c3.reale; // Illegale
 }
}
```

LP1 – Lezione 12

6 / 39

### private (2)

- 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) {
 reale = r; // Illegale
 }
}
```

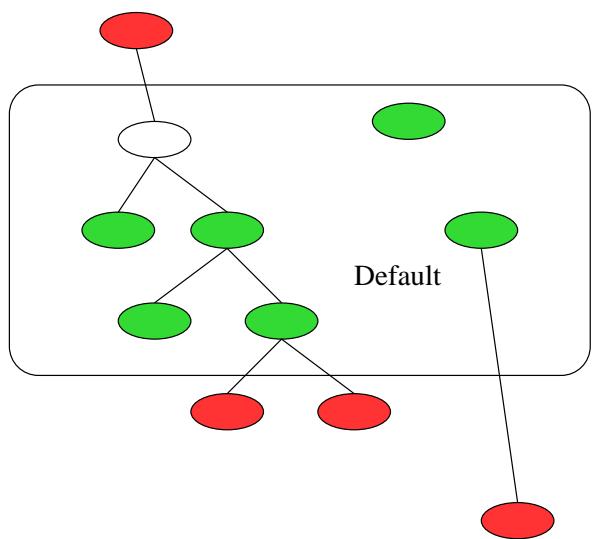
- La classe `SubComplesso` eredita gli attributi della superclasse, MA quegli attributi possono essere usati solo dal codice della classe `Complesso`.
- Per quanto riguarda la classe `SubComplesso`, è come se quegli attributi non li avesse ereditati affatto.
- Tutto quanto detto è valido anche per metodi privati; i costruttori, invece, non sono mai ereditati.

LP1 – Lezione 12

7 / 39

## 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 pacchetto.

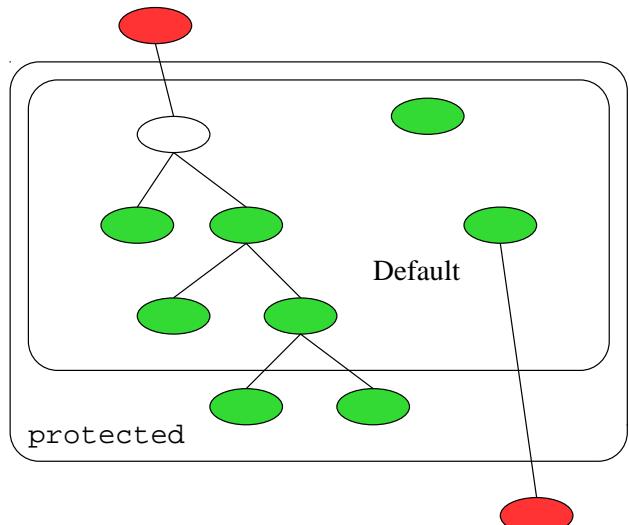


LP1 – Lezione 12

8 / 39

## protected (1)

- Accesso consentito a tutte le classi nello stesso pacchetto e
- a tutte le sottoclassi in pacchetti diversi, ma solo per ereditarietà: una sottoclasse in un altro pacchetto può accedere ad un membro protected nella superclasse 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.



LP1 – Lezione 12

9 / 39

## protected (2)

Esempio:

```
package p1;

public class A {
 protected int x = 7;
}
```

```
package p2;
import p1.A;

public class B extends A {
 public void testProt() {
 System.out.println(x); // Ok: x e' ereditato
 // uso implicito di this
 A a = new A();
 System.out.println(a.x); // Errore di comp.
 // a non e' di tipo B, ne' di sottotipo di B.
 B b = new B();
 System.out.println(b.x); // Ok: accesso
 // attraverso oggetto di tipo B.
 }
}
```

## Overriding

Lo dico ancora:

- I metodi visibili di una superclasse non possono essere sovrapposti in una sottoclasse da altri meno visibili.
- **I metodi visibili di una superclasse non possono essere sovrapposti in una sottoclasse da altri meno visibili.**
- **I metodi visibili di una superclasse non possono essere sovrapposti in una sottoclasse da altri meno visibili.**

### 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. Attenzione: 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, perché esso non è mai ereditato dalle sottoclassi, quindi mai sovrapponibile.

LP1 – Lezione 12

13 / 39

### abstract

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

```
abstract void getValore();
```
- Una classe PUÒ essere marcata **abstract**. In questo caso il compilatore suppone che essa 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 [vedremo in seguito], 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.

LP1 – Lezione 12

14 / 39

### 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 zero) di istanze di quella classe.

LP1 – Lezione 12

15 / 39

## - 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 (perché potrebbero anche non esistere).
- Si veda il metodo main.

```
class Boo {
 static int i = 48;
 int j = 1;

 public static void main (String args[]) {
 i += 100;
 // j *= 5; Per fortuna e' 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++);
 }
}
```

LP1 – Lezione 12

18 / 39

## 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.

LP1 – Lezione 12

19 / 39

## transient

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

LP1 – Lezione 12

20 / 39

## synchronized

- Si applica solo a metodi o a blocchi anonimi.
- Controlla l'accesso al codice in programmi multi-thread.

LP1 – Lezione 12

21 / 39

## volatile

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

LP1 – Lezione 12

22 / 39

## Sommario

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

LP1 – Lezione 12

23 / 39

## Esempio

24 / 39

### Forme ricorrenti

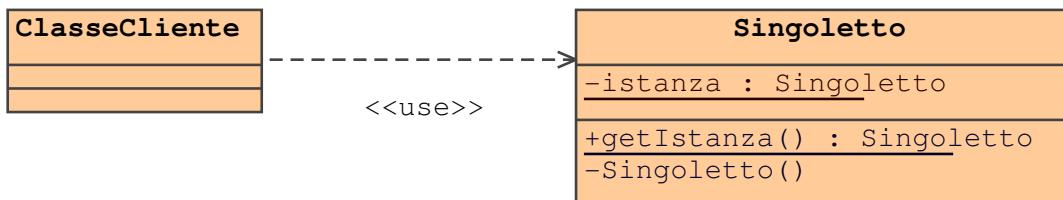
- *Forme progettuali ricorrenti* (Design Patterns) sono soluzioni a problemi ricorrenti nella progettazione OO.
- Visitare <http://hillside.net/patterns> per maggiori informazioni.

LP1 – Lezione 12

25 / 39

### Singololetto (1)

- Uno dei problemi ricorrenti è il seguente: si vuole che una classe abbia una sola istanza.
- Se si scrivesse la classe nel modo solito, gli utilizzatori potrebbero creare istanze a volontà.
- Lo scopo della forma progettuale *Singleton* è quello di assicurare il progettista della classe che esisterà sempre una ed una sola istanza di quella classe.
- La forma è la seguente:

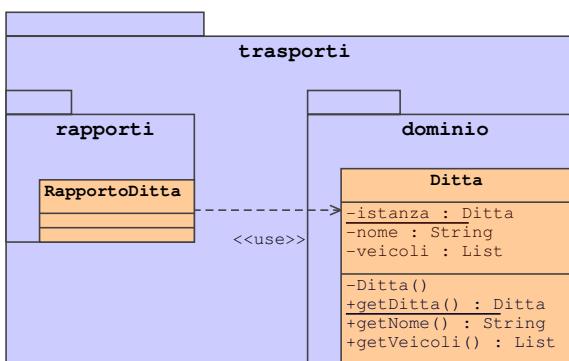


Nota: In UML le componenti static in una classe sono sottolineate. L'associazione tratteggiata <<use>> descrive non un attributo della ClasseCliente di tipo Singololetto, ma un uso diretto della classe Singololetto.

LP1 – Lezione 12

26 / 39

## Singololetto (2)



```
package trasporti.rapporti;

import trasporti.dominio.*;

public class RapportoDitta {
 public void generaTesto
 (PrintStream output) {
 Ditta d = Ditta.getDitta();
```

```
// usa l'oggetto Ditta per
// reperire e stampare le
// info della Ditta.
}
```

```
package trasporti.dominio;

public class Ditta {
 private static Ditta istanza =
 new Ditta();
 private String nome;
 private Veicolo[] flotta;

 public static Ditta getDitta() {
 return istanza;
 }

 private Ditta() {...}

 // ulteriore codice
}
```

LP1 – Lezione 12

27 / 39

## Esercizi

28 / 39

### Esercizi

1. Modificare la classe Banca in modo che realizzi un singololetto.
2. È possibile modificare la visibilità `protected` di `saldo` nella classe Conto in modo che sia nuovamente `private` e ripristinare così l'incapsulazione perfetta?  
La risposta è sì. Ma questo ha conseguenze nel codice della classe ContoCorrente.  
Esibire le necessarie (ed eleganti) modifiche nella classe ContoCorrente che rendano nuovamente il programma funzionante.

LP1 – Lezione 12

28 / 39

**D 1**

Quale dei seguenti frammenti viene correttamente compilato e stampa "Uguale" in esecuzione?

A. 

```
Integer x = new Integer(100);
Integer y = new Integer(100);
if (x == y) {
 System.out.println("Uguale");
}
```

B. 

```
int x=100;
Integer y = new Integer(100);
if (x == y) {
 System.out.println("Uguale");
}
```

C. 

```
int x=100; float y=100.0F;
if (x == y) {
 System.out.println("Uguale");
}
```

D. 

```
String x = new String("100");
String y = new String("100");
if (x == y) {
 System.out.println("Uguale");
}
```

E. 

```
String x = "100";
String y = "100";
if (x == y) {
 System.out.println("Uguale");
}
```

**D 2**

Quali delle seguenti dichiarazioni sono illegali?

- A. default String s;
- B. transient int i = 41;
- C. public final static native int w();
- D. abstract double d;
- E. abstract final double cosenoIperbolico();

**D 3**

Quale delle seguenti affermazioni è vera?

- A. Una classe abstract non può avere metodi final.
- B. Una classe final non può avere metodi abstract.

## D 4

Qual è la minima modifica che rende il seguente codice compilabile?

```
1. final class Aaa {
2. int xxx;
3. void yyy() { xxx=1; }
4. }
5.
6. class Bbb extends Aaa {
7. final Aaa fref = new Aaa();
8. final void yyy() {
9. System.out.println(
10. "In yyy()");
11. fref.xxx = 12345;
12. }
13. }
```

- A. Alla linea 1, rimuovere `final`.
- B. Alla linea 7, rimuovere `final`.
- C. Rimuovere la linea 11.
- D. Alle linee 1 e 7, rimuovere `final`.
- E. Nessuna modifica è necessaria.

LP1 – Lezione 12

33 / 39

## D 5

Riguardo al codice seguente, quale affermazione è vera?

```
1. class Roba {
2. static int x = 10;
3. static { x += 5; }
4.
5. public static void main(String[] args) {
6. System.out.println("x=" + x);
7. }
8.
9. static { x /= 5; }
10. }
```

- A. Le linee 3 e 9 non sono compilate, poiché mancano i nomi di metodi e i tipi di ritorno.
- B. La linea 9 non è compilata, poiché si può avere solo un blocco top-level `static`.
- C. Il codice viene compilato e l'esecuzione produce `x=10`.
- D. Il codice viene compilato e l'esecuzione produce `x=15`.
- E. Il codice viene compilato e l'esecuzione produce `x=3`.

LP1 – Lezione 12

34 / 39

## D 6

Rispetto al codice seguente, quale affermazione è vera?

```
1. class A {
2. private static int x=100;
3.
4. public static void main(
5. String[] args); {
6. A hs1 = new A();
7. hs1.x++;
8. A hs2 = new A();
9. hs2.x++;
10. hs1 = new A();
11. hs1.x++;
12. A.x++;
13. System.out.println(
14. "x = " + x);
```

```
15. }
16. }
```

- A. La linea 7 non compila, poiché è un riferimento static ad una variabile private.
- B. La linea 12 non compila, poiché è un riferimento static ad una variabile private.
- C. Il programma viene compilato e stampa x = 102.
- D. Il programma viene compilato e stampa x = 103.
- E. Il programma viene compilato e stampa x = 104.

LP1 – Lezione 12

35 / 39

## D 7

Dato il codice seguente:

```
1. class SuperC {
2. void unMetodo() { }
3. }
4.
5. class SubC extends SuperC {
6. void unMetodo() { }
7. }
```

- 1. Quali modificatori di accesso possono essere legalmente dati ad unMetodo alla linea 2, lasciando il resto del codice inalterato?
- 2. Quali modificatori di accesso possono essere legalmente dati ad unMetodo alla linea 6, lasciando il resto del codice inalterato?

LP1 – Lezione 12

36 / 39

## D 8

```
1. package abcd;
2.
3. public class SupA {
4. protected static int count=0;
5. public SupA() { count++; }
6. protected void f() {}
7. static int getCount() {
8. return count;
9. }
10. }
```

```
8. }
```

Riguardo ai codici a sinistra, quale affermazione è vera?

- A. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e A è nello stesso pacchetto di SupA.
- B. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e public nella sottoclasse.
- C. La compilazione di A.java fallisce alla linea 5, poiché il metodo getCount() è static nella superclasse e non può essere sovrapposto da un metodo non-static.
- D. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo f() su una istanza di A.
- E. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo getCount su una istanza di SupA.

```
1. package abcd;
2.
3. class A extends abcd.SupA {
4. public void f() {}
5. public int getCount() {
6. return count;
7. } }
```

LP1 – Lezione 12

37 / 39

## D 9

Riguardo ai codici seguenti, quale affermazione è vera?

```
1. package abcd;
2.
3. public class SupA {
4. protected static int count=0;
5. public SupA() { count++; }
6. protected void f() {}
7. static int getCount() {
8. return count;
9. }
10.}
```

```
7. String [] args) {
8. System.out.print(
9. "Prima:" + count);
10. A a = new A();
11. System.out.println(
12. " Dopo:" + count);
13. a.f();
14. }
15.}
```

```
1. package ab;
2.
3. class A extends abcd.SupA {
4. A() { count++; }
5.
6. public static void main(
```

- A. Il programma viene compilato e stampa: Prima:0 Dopo:2
- B. Il programma viene compilato e stampa: Prima:0 Dopo:1
- C. La compilazione di A fallisce alla linea 4.
- D. La compilazione di A fallisce alla linea 13.
- E. Il programma viene compilato, ma viene lanciata una eccezione alla linea 13.

## D 10

Si consideri la classe seguente:

```
1. public class Test1 {
2. public float unMetodo(float a, float b) {
3. }
4.
5. }
```

Quali dei seguenti metodi può lecitamente essere inserito alla linea 4?

- A. public int unMetodo(int a, int b) { }
- B. public float unMetodo(float a, float b) { }
- C. public float unMetodo(float a, float b, int c) { }
- D. public float unMetodo(float c, float d) { }
- E. private float unMetodo(int a, int b, int c) { }

---

# Linguaggi di Programmazione I – Lezione 17

Prof. Piero Bonatti  
<mailto://pab@unina.it>

26 gennaio 2020



# Panoramica dell'argomento

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



## Paradigma funzionale

Pensare funzionale

Esempi

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

# Paradigma funzionale



# L'essenza del paradigma funzionale

Paradigma funzionale

Pensare funzionale

Esempi

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive
- Non vi sono assegnamenti, non c'è una memoria che cambia
  - ◆ perché 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 ci possono essere cicli while/for
- Conseguenze sulla programmazione:
  - ◆ ricorsione al posto dei cicli
  - ◆ modifiche all'ambiente anzichè agli assegnamenti:
  - ◆ creazione di nuovi identificatori con lo stesso nome che mascherano la versione precedente (come nello scoping statico)



## Esempi

- In questo corso illustreremo brevemente due linguaggi funzionali:
  - ◆ ML (nella versione Standard ML of New Jersey)
  - ◆ Scheme
- Entrambi i linguaggi supportano le seguenti implementazioni:
  - ◆ interprete interattivo (lo useremo per molti esempi)
  - ◆ implementazione mista (compilazione su codice intermedio e successiva interpretazione)
  - ◆ compilazione su codice oggetto *standalone*
- Testi di riferimento
  - ◆ **ML**: Riccardo Pucella. *Notes on Programming Standard ML of New Jersey*.  
<https://www.cs.cornell.edu/riccardo/prog-smlnj/notes-011001.pdf>



## Paradigma funzionale

### **ML**

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# ML



## Il sistema di tipi

- ML è *fortemente e staticamente tipato*
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
  - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)
- Supporta *polimorfismo parametrico* (come i template)
- Ha un *garbage collector*
- Supporta *encapsulation* (tipi di dato astratti) ma non è un linguaggio a oggetti
  - ◆ mancano la gerarchia di tipi e – di conseguenza – l'ereditarietà
- Il linguaggio OCaml supporta anche gerarchie di tipi ed ereditarietà (è object-oriented)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Implementazioni di ML

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

ML può essere usato in 2 modi:

1. Interagendo con l'interprete (ad es. **Standard ML of New Jersey**)

- 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 utilizzabili le dichiarazioni contenute nel file

2. Compilando un programma in codice oggetto direttamente eseguibile

- ad es. mediante il compilatore **mlton** per standard ML

```
mlton "nome del file.sml"
```

questo comando produce un file eseguibile con lo stesso nome (ma senza l'estensione .sml)



# I tipi primitivi (I)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Detti anche *base types*

## ■ **int**: gli interi

0, 1, ~1, 2, ~2, ...      0xff, ~0x32, ...

Notare che si usa ~ invece del segno meno (-). Alcuni operatori:

+, -, \*, div, mod, =, <, ...

## ■ **word**: unsigned integers

0w44, ~0w15, 0wxff, ...

## ■ **real**

3.14, 2.0, 0.1E6, ...

Alcuni operatori su real:

+, -, \*, /, =, <, ...



# I tipi primitivi (II)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ string

```
"abc", "123", ...
```

Alcuni operatori su string:

```
^, size, =, <, ...
```

## ■ char

```
#"a", #"\n", #"\163", ...
```

Alcuni operatori su real:

```
ord, chr, =, <, ...
```

## ■ bool

```
true, false
```

Alcuni operatori su bool:

```
not, andalso, orelse, =
```



# Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- 0w7 mod 0w4;
val it = 0wx3 : word

- "Hallo " ^ "world";
val it = "Hallo world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int

- 3 + 2.2;
Error: operator and operand don't agree [overload conflict]

- real(3) + 2.2;
val it = 5.2 : real
```

## ■ 'it' si riferisce all'espressione data; calcola sia valore che tipo



# I tipi primitivi (III)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- *Nessuna conversione automatica tra tipi numerici! Usare real:int->real e basis library*

```
- val r = 3.0 + 2;
Error: operator and operand don't agree

- val r = 3.0 + real(2);
val r = 5.0 : real

- val i = 1 + 0w1;
Error: operator and operand don't agree

- val i = 1 + Word.toInt(0w1);
val i = 2 : int
```

- C'è una basis library per ogni tipo primitivo (Int, Word, Real...) con funzioni per conversioni, parsing, e altre utilità



## I tipi primitivi (IV)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Real non supporta l'uguaglianza! Usare Real.==

```
- val x = 1.0; val y = 2.0;
val x = 1.0 : real
val y = 2.0 : real
- x = y;
Error: operator and operand don't agree [equality type required]

- Real.==(x,y);
val it = false : bool
```

- Questo perchè lo standard IEEE prevede valori che risultano da operazioni non definite, denominati **NaN** (not a number)

- ◆ Un NaN non è confrontabile con nessun altro numero, nemmeno con sè stessi

```
- val e = Math.sqrt(~2.0);
val e = nan : real

- Real.==(e,e);
val it = false : bool
```



Paradigma  
funzionale

---

ML

---

Dichiarazioni e  
scoping in ML

Funzioni

Altre dichiarazioni di  
identificatori

Scoping

Tipi strutturati in  
ML

---

Patterns e matching

---

Liste

---

Currying

---

Funzioni di ordine  
superiore

---

Polimorfismo  
parametrico

---

Encapsulation e  
interfacce

---

Eccezioni e  
integrazione con  
type checking

---

Esempio: un  
semplice compilatore

---

# Dichiarazioni e scoping in ML



# Funzioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ci sono diversi modi di definire e chiamare funzioni in ML.  
Iniziamo con i più tradizionali

```
- fun fatt x = if x=0 then 1 else x*fatt(x-1);
val fatt = fn : int -> int

- fatt(3);
val it = 6 : int

- fatt 3; (* in questo caso le parentesi sono opzionali *)
val it = 6 : int
```

- Con **fun** si *dichiara* la funzione
  - ◆ fun aggiunge all'ambiente l'identificatore **fatt**
  - ◆ e lo associa alla funzione da interi a interi
- Si può vedere cosa è associato a **fatt** senza *chiamare la funzione*

```
- fatt; (* nome della funzione senza argomenti *)
val it = fn : int -> int
```

Mostra solo il tipo (il valore è stato trasformato in bytecode)



# Altre dichiarazioni di identificatori

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Con **val** si aggiunge un nuovo identificatore all'ambiente e gli si associa un valore

```
- val x = 2+2;
val x = 4 : int
```

```
- x+2;
val it = 6 : int
```

- Grammatica delle dichiarazioni viste sinora

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

Vedremo più avanti che **val** è più generale di **fun**



# Scoping

- L'equivalente dei blocchi in ML è

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

Lo scoping è **statico**. Esempi:

```
- let val x=2 in 3*x end;
val it = 6 : int

- x;
Error: unbound variable or constructor: x

- let val x=2 in
 let val x=3 in (* questa def. maschera la precedente *)
 3*x
 end
 end;
val it = 9 : int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni  
Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Scoping (II)

## ■ Ambiente non locale delle funzioni

```
- val x=0;
val x=0 : int

- let val x=1 in
 let fun f(y) = x+y in (* x è non locale *)
 f(0)
 end
end;
val it = 1 : int
```

## ■ Forma equivalente più concisa

```
let
 val x=1
 fun f(y) = x+y
in
 f(0)
end;
```

dopo “let” possiamo mettere quante dichiarazioni vogliamo

Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Funzioni  
Altre dichiarazioni di  
identificatori

Scoping

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore



## Scoping (III)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni  
Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

### Definizioni ausiliarie

- locali ad altre definizioni

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

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



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e  
costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# Tipi strutturati in ML



# Prodotti cartesiani

- Si possono definire  $n$ -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘\*’
- Si estrae l’ $i$ -esimo elemento da una  $n$ -upla con  $\#i$

```
- (1+1, "A");
val it = (2,"A") : int * string

- val x = (1,"A",3.5);
val x = (1,"A",3.5) : int * string * real

- #1(x);
val it = 1 : int

- #2(x);
val it = "A" : string

- #3(x);
val it = 3.5 : real
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

**Prodotti cartesiani**

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Record

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Insiemi di espressioni `<nome>=<valore>`. Notate come viene rappresentato il tipo

```
- val r = {nome="Mario",nato=1998};
val r = {nato=1998, nome="Mario"} : {nato:int, nome:string}
```

- Il valore associato al nome  $N$  si estrae con `#N`

```
- #nome(r);
val it = "Mario" : string

- #nato(r);
val it = 1998 : int
```

- L'ordine delle coppie non conta

```
- {nome="Mario", nato=1998} = {nato=1998, nome="Mario"};
val it = true : bool
```



# Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai `typedef` del C

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

- Il compilatore va aiutato a stabilire il tipo

```
- val x = (3.0,4.0); (* senza aiutino *)
val x = (3.0,4.0) : real * real

- val x:coord = (3.0,4.0); (* con aiutino *)
val x = (3.0,4.0) : coord
```

- I tipi `coord` e `(real * real)` sono compatibili tra loro (*structural equivalence*)

- ◆ posso passare una espressione di tipo `coord` a un parametro di tipo `(real * real)` e viceversa
- ◆ similmente `coord` è compatibile con ogni altro tipo definito come `(real * real)`, come ad esempio

```
type coppia = real * real;
```



# Datatypes e costruttori

- Con **datatype** si può fare di più che dare un nome a un tipo ML

- ◆ si possono definire **costruttori** per creare *data objects*

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

- val c = red;
val c = red : color
```

red, green, blue sono costruttori. Definiscono i possibili valori del tipo color

- Notare la somiglianza con le enum del C. Solo apparente...

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };
printf("%d%d%d", red,green,blue); (* stampa 012 *)

enum color c = red;
if(c == 0) then ... else ...; (* esegue il then *)

c = 10; (* nessun errore!!! *)
```

- Invece i datatypes di ML definiscono tipi genuinamente nuovi: nessuna corrispondenza con gli int

```
- val c:color = 10;
Error: pattern and expression in val dec don't agree

- c = 0; (* c è uguale a 0? *)
Error: operator and operand don't agree
```

red, green, blue sono oggetti completamente nuovi

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



# Costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

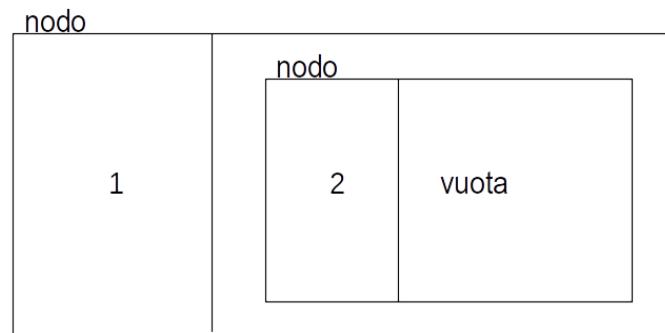
Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
  - ◆ la lista vuota (caso base)
  - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)



Rappresentazione della lista [1,2]

- Quindi servono 2 costruttori: per la lista vuota e per i nodi

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

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



# Costruttori con argomenti (II)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Altro esempio: albero binario con nodi etichettati da interi

■ Definizione ricorsiva: un albero simile è

- ◆ un albero vuoto, oppure
- ◆ un nodo che contiene un intero e due alberi dello stesso tipo

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

In questo esempio costruiamo un albero con radice 1, figlio sinistro 2 (che è una foglia), mentre il figlio destro manca.

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



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# Patterns e matching



# Utilizzo dei costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota

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

- L = vuota;
val it = false : bool
```

- Se non è vuota potrebbe servirci il primo elemento. Si estrae con *pattern matching*

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

La parte in rosso è chiamata *pattern*

- Per ottenere il resto della lista

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



# Funzione che conta gli elementi della lista

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Ovviamente deve essere *ricorsiva* (niente cicli!)

```
- fun conta x = if x=vuota then 0
 else let val nodo(_, r) = x
 in conta(r) + 1 end;

val conta = fn : listaInt -> int

- conta L;
val it = 2 : int
```

## ■ Notare come il compilatore ha *inferito* il tipo della funzione

1. 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
2. il “then” restituisce 0, che è un intero; quindi l’output di “conta” è un intero

## ■ Inoltre il compilatore controlla che anche il resto della funzione sia compatibile con questi tipi

1. r corrisponde al 2<sup>o</sup> argomento del nodo, che è di tipo listaInt  $\Rightarrow$  è corretto passarlo a conta che restituisce un intero
2. quindi anche l’else restituisce un intero  $\Rightarrow$  tutto torna



# Abbreviazioni per i pattern

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

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

```
- val nodo(p, r) = L;
val p = 1 : int
val r = nodo (2, vuota) : listaInt
```

Cioè dichiara due identificatori (p e r) in un colpo solo

- Si può *definire una funzione per casi*

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

mettendo direttamente i pattern al posto dei parametri formali

Notare l'eleganza e la concisione



# Analogo per i pattern dello switch/case del C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- L'idea è la stessa della definizione per casi delle funzioni

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

val it = false : bool
```



## Esercizi

1. Scrivere in ML una funzione che dato un albero binario A come quello visto in precedenza, e dato un intero N, restituisce l'etichetta del nodo che si raggiunge in N passi visitando l'albero in preordine
2. Simile all'esercizio 1, ma visitando l'albero in postordine (difficile)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# Liste



# Le liste in ML

- Le liste sono tra le strutture dati più usate in programmazione funzionale

- ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)

- Perciò ML le fornisce built-in con i costruttori **nil** e **::**

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

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

(* sono veramente equivalenti *)
- [] = nil;
val it = true : bool

- 1::2::3::nil = [1,2,3];
val it = true : bool
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Principali operatori sulle liste in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La funzione `length` restituisce la lunghezza di una stringa
- La funzione `null` restituisce true se la stringa è vuota
- Le funzioni `hd` e `tail` restituiscono il primo elemento e il resto della lista, rispettivamente

```
- val L = [1,2,3];
val L = [1,2,3] : int list

- hd L;
val it = 1 : int

- tl L;
val it = [2,3] : int list
```

- Altre funzioni si trovano nella struttura `List`, ad esempio
    - ◆ `List.nth(L,i)` restituisce l' $i$ -esimo elemento di  $L$  (partendo da 0)
    - ◆ `List.last(L)` restituisce l'ultimo elemento di  $L$
- vedere <http://sml-family.org/Basis/list.html>



# Esercizi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Scrivere in ML:

1. le funzioni standard su liste riportate nella slide precedente;
2. una funzione `member` che dati una lista L e un intero N, restituisce `true` se e solo se N appartiene ad L.
3. una funzione che dati una lista L e un intero N, appende N in fondo alla lista;
4. una funzione che concatena due liste
5. una funzione `reverse` che inverte l'ordine degli elementi di una lista (suggerimento: usare un parametro aggiuntivo che serve a costruire progressivamente la lista invertita)



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# Currying



# Funzioni con più argomenti: esistono?

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- In realtà 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 '='

- Esempio semplice

```
- 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  $\text{int} \rightarrow \text{int} \rightarrow \text{int}$  va inteso come  $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$



# Currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La trasformazione da n-uple (come  $f(x,y)$ ) a funzioni che restituiscono funzioni (come  $f' x y$ ) si chiama *currying*
- L'utilizzo è ovviamente diverso

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

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

- f (3,2);
val it = 5 : int

- f' 3 2; (* viene interpretato come (f'3)(2) *)
val it = 5 : int

- f 3 2;
Error: operator and operand don't agree

- f' (3,2);
Error: operator and operand don't agree

- val g = f' 3;
val g = fn : int -> int

- g 1;
val it = 4 : int
```



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# Funzioni di ordine superiore



# Invece dei cicli: Funzioni di ordine superiore

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- 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 volta per tutte 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 tre:
  - ◆ filter
  - ◆ map
  - ◆ reduce

Nel seguito mostriamo le loro versioni per le liste



# Filter

- La funzione filter prende una funzione booleana f e una lista L
- 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
```

Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore



# Map

- La funzione `map` prende una funzione booleana `f` e una lista `L`
- applica `f` a tutti gli elementi della lista

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

(* esempio: conversione in lista di reali *)
- map real [1,2,3];
val it = [1.0,2.0,3.0] : real list

(* esempio: conversione in lista di stringhe *)
- map Int.toString [1,2,3];
val it = ["1","2","3"] : string list
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

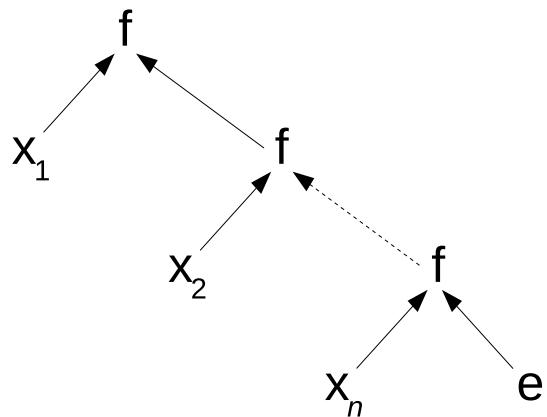
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



## 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  $e$  e una lista  $L$  ed effettua questo calcolo:

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


Ad esempio se  $f$  è '+' ed  $e=0$  allora `reduce` calcola la somma degli elementi della lista

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Reduce

## ■ Definizione di reduce ed esempi

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

(* esempio: somma (sbagliato, + è infisso...) *)
- reduce + 0 [1,2,3];
Error: ...

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

(* esempio: media di $[x_1, \dots, x_n] = x_1/n + \dots + x_n/n$ *)
- let
 fun f L (elem, accum) = elem / real(length L) + accum
 val lista = [1.0,2.0,3.0]
 in
 reduce (f lista) 0.0 lista
 end;
val it = 2.0 : real
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



## Esercizi

1. Implementare in ML le seguenti funzioni, sfruttando al meglio filter, map e reduce
  - (a) pari: data una lista di interi L, calcolare la sottolista che contiene solo i numeri pari in L
  - (b) membro: dati una lista di interi e un intero, restituisce *true* se e solo se l'intero compare nella lista
  - (c) conversione: data una lista di interi convertirla in una lista di reali

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Filter,Map,Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Funzioni anonymous

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter, Map, Reduce

Funzioni anonymous

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Quando utilizziamo funzioni di ordine superiori come filter, map e reduce, può far comodo passargli funzioni semplici, specificate lì per lì senza dover dare loro un nome (nè usare un blocco *let*)
- Queste funzioni anonymous 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 lambda calcolo l'operatore  $\lambda$  è l'analogo di fn
    - come in  $\lambda x. x + 1$



# Esercizi su funzioni anonime

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

1. Spiegare perchè una funzione anonima non può essere ricorsiva
2. Svolgere il seguente esercizio usando al meglio una delle funzioni filter, map, reduce e *una opportuna funzione anonima*
  - (a) pari: data una lista di interi L, calcolare la sottolista che contiene solo i numeri pari in L



## Val vs. fun

- Adesso possiamo vedere in che senso val è più generale di fun. Le seguenti definizioni sono equivalenti:

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

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

In altre parole, fun è *zucchero sintattico*, cioè una utile abbreviazione per qualcosa che si potrebbe fare in altro modo (con val)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Ulteriori dettagli su currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter, Map, Reduce  
Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Adesso possiamo mostrare più esplicitamente la natura del currying. Riprendiamo l'esempio

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

- In effetti  $f'$  può essere definita equivalentemente come

```
fun f' x = fn y => x+y
```

```
val f' = fn : int -> int -> int
```

- L'esempio della chiamata di  $f'$  con un solo parametro può essere spiegata così:

```
- val g = f' 3; (* come fosse val g = fn y => 3+y *)
val g = fn : int -> int
```

```
- g 1;
val it = 4 : int
```



# Ulteriori dettagli su currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- È anche possibile definire funzioni che effettuano il currying 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
```

- Esempi di utilizzo

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

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

oppure

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

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



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Tipi parametrici

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# Polimorfismo parametrico



# Tipi parametrici

- ML supporta l'analogo dei *template* di C++ e Java, ovvero *tipi parametrici*
- In realtà li stiamo usando da quando usiamo le liste:
  - ◆ il costruttore :: può essere applicato a qualunque tipo

```
1 :: nil oppure "abc" :: nil ...
- length;
val it = fn : 'a list -> int
```

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

- ◆ cioè accetta liste con qualsiasi contenuto
- ◆ in effetti non ha bisogno di saperne il tipo: deve solo contare in nodi
- Anche la funzione map è parametrica:

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```

Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Tipi parametrici

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore



# Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Come definire tipi parametrici (come le liste )

```
(* generalizzazione delle nostre liste *)
```

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

```
(* alberi binari con etichette parametriche *)
```

```
- 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 la *type inference*

```
- fun conta(vuota) = 0
 | conta (nodo(x,l)) = conta(l) + 1;
val conta = fn : 'a lista -> int
```



# Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*
  - ◆ ad esempio '**a** o '**b** o '**c** ...
- Quando si vuole che il tipo supporti l'uguaglianza, allora si mette un doppio apice
  - ◆ ad esempio "**a** o "**b** o "**c** ...
- Ogni tanto la type inference se ne accorge da sola

```
- fun diag (x,y) = x=y;
val diag = fn : "a * "a -> bool

- diag (1.0, 1.0);
Error: operator and operand don't agree [equality type required]
operator domain: ''Z * ''Z
operand: real * real
```

(ricordarsi che i reali non supportano l'uguaglianza...)



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[\*\*Encapsulation e interfacce\*\*](#)

[Signatures](#)

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Encapsulation e interfacce



# Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: 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 com'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
- ...senza dire come sono implementate (ovviamente)...
- *per assegnare un tipo a una espressione usare :*



# Structures = Implementazioni delle signature

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

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

- Con l'espressione Stack :> STACK diciamo diverse cose:
  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 dichiarate in STACK
    - ◆ ogni altra funzione definita nella structure non è accessibile da fuori
    - ◆ così si ottiene l'*encapsulation*
    - ◆ e si definiscono *tipi di dato astratti*



# Incapsulamento dell'implementazione dei tipi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Anche se in Stack il tipo stack è implementato con list...

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

- ... non si può usare come list ...

```
- length [];
 val it = 0 : int

- length Stack.empty;
 stdIn:39.1-39.19 Error: operator and operand don't agree
 operator domain: 'Z list
 operand: 'Y Stack.stack
```

- ... perchè la structure Stack non mette a disposizione alcuna funzione length sul tipo stack e ne nasconde l'implementazione



# Functors = structure parametriche

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri
- Si usa una keyword diversa: **functor**. Analogo dei template
- Ecco un esempio di definizione di immagini parametrica rispetto alla codifica del colore
  - ◆ supponiamo di avere due structure RGB e CMYK per gli omonimi modelli di colore
  - ◆ e che entrambe implementino la signature COLOR
  - ◆ la struttura parametrica si può dichiarare così:

```
functor Image(X : COLOR) =
 struct
 (* qui si può usare X come un tipo *)
 (* con le operazioni definite da COLOR *)
 end

 (* col functor si possono generare diversi tipi di dato *)
 structure Image_RGB = Image(RGB);
 structure Image_CMYK = Image(CMYK);
```



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

# Eccezioni e integrazione con type checking



# Eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col 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 due costruttori: `::` e `[]`
3. la definizione per casi di pop ha un caso solo per `::`
4. da cui il warning
5. il compilatore inserisce automaticamente una eccezione `Match` nei casi mancanti



# Dichiarazione e generazione eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Il programmatore può definire le proprie eccezioni:

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

fun pop(x::s) = (x,s)
| pop [] = raise EmptyStack; (* come il throw di Java *)
```

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

```
- pop [];
uncaught exception EmptyStack
```

- Le eccezioni possono essere catturate e gestite con handle:

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



## Il costrutto handle

- può essere messo dopo qualunque espressione che può generare una eccezione

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

- un singolo handle può gestire diverse eccezioni

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

*Ogni ordine è buono: perchè?*

- Due modi di usarlo in ML funzionale puro:

1. fare qualcosa come stampare un messaggio e *rilanciare l'eccezione*
2. “aggiustare” l’errore restituendo un valore *dello stesso tipo* dell’espressione che ha sollevato l’eccezione

Sono le uniche opzioni che passano il type checking senza errori

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



## Il costrutto handle

### ■ Altro esempio (da non seguire acriticamente ...)

```
- fun pos x (y::z) =
 if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])

### ■ Si può usare handle per modificare pos per restituire -1 quando non trova x nella lista:

```
- fun pos2 x y = (pos x y) handle Match => ~1;
```

### ■ Esempio didattico un po' artificiale: si potrebbe obiettare che pos è realizzata male...

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Eccezioni con parametri

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni
- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```

- Questa eccezione può essere lanciata in diversi modi...

```
raise SyntaxError "Identifier expected"
```

```
raise SyntaxError "Integer expected"
```

- ... e il parametro “letto” col pattern matching

```
... handle SyntaxError x => ... (* qui si può usare x *)
```



Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

## Esempio: compilazione di espressioni



# Elaborazione di simboli in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML – come gli altri linguaggi dichiarativi (= non imperativi) – è particolarmente adatto alla *manipolazione di simboli*
- La *realizzazione di compilatori* è un esempio di questo tipo di problema
  - ◆ bisogna elaborare espressioni e comandi di un linguaggio di programmazione (codice sorgente) e tradurli in un altro linguaggio (codice oggetto o intermedio)
- Ne approfittiamo per dare un'idea parziale di alcune strutture dati interne al compilatore e dei procedimenti di generazione e ottimizzazione del codice
- L'esempio che segue realizza un compilatore per un linguaggio molto semplificato che supporta solo semplici espressioni su numeri interi. Il codice oggetto deve calcolare l'espressione data.



# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

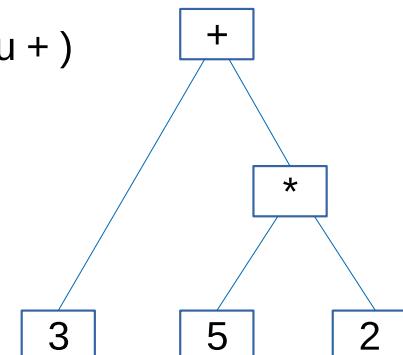
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$



Albero sintattico  
(\* ha precedenza su +)



Procedimento di calcolo:

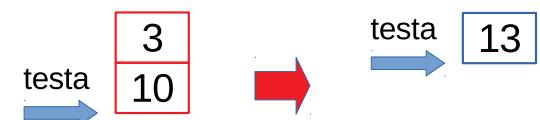
- salvare i risultati intermedi (come  $5 * 2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 \* +



Stack



*Il compilatore, a partire dall'albero sintattico, deve generare un codice che implementa questo procedimento*



# Definizione degli alberi sintattici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Introduciamo un costruttore per ogni operazione supportata dal linguaggio sorgente
- ogni costruttore corrisponde a un tipo di nodo dell'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
```



# Definizione del linguaggio target

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
 = LOADC of int * int (* LOADC i c => Ri := c *)
 | LOADI of int * int (* LOADI i j => Ri := mem(Rj) *)
 | STOREI of int * int (* STOREI i j => mem(Rj) := Ri *)
 | INCR of int (* INCR i => Ri := Ri + 1 *)
 | DECR of int (* DECR i => Ri := Ri - 1 *)
 | SUM of int * int (* SUM i j => Ri := Ri + Rj *)
 | SUB of int * int (* SUB i j => Ri := Ri - Rj *)
 | MUL of int * int (* MUL i j => Ri := Ri * Rj *)
 | DIV of int * int (* DIV i j => Ri := Ri / Rj *)
 | MOD of int * int (* MOD i j => Ri := Ri mod Rj *)
 | HALT
```



# Generazione del codice – funzione codegen

Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

- Il codice oggetto utilizza due registri:
  - ◆ R1 come puntatore alla testa dello stack
  - ◆ R2 come accumulatore (per calcolare le singole operazioni)
- 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* avere 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]



# Generazione del codice

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
fun codegen tree =
let
 (* definizione della funzione translate *)
 (* R1: stack pointer; R2: accumulator *)
 fun translate (co x) cont =
 LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont
 | translate (plus(t1,t2)) cont =
 translate t1 (
 translate t2 (
 LOADI(2,1)::DECR(1)::LOADI(3,1)::SUM(2,3)::STOREI(2,1)::cont))
 | translate (times(t1,t2)) cont = simile ma con MUL al posto di SUM
 | translate (minus(t1,t2)) cont = simile ma con SUB(3,2) al posto di SUM(2,3)
 | translate (divide(t1,t2)) cont = simile ma con DIV al posto di SUB
 | translate (modulus(t1,t2)) cont = simile ma con MOD al posto di SUB
in
 translate tree [HALT]
end
```



# Ottimizzazione del codice

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La generazione meccanica introduce operazioni inutili.

```
LOADC 2 3 (* codice generato per 3+5*2 *)
```

```
INCR 1
```

```
STOREI 2 1
```

—

```
LOADC 2 5
```

```
INCR 1
```

```
STOREI 2 1
```

—

```
LOADC 2 2
```

```
INCR 1
```

```
STOREI 2 1
```

—

```
LOADI 2 1
```

```
DECR 1
```

```
LOADI 3 1
```

```
MUL 2 3
```

```
STOREI 2 1
```

—

```
LOADI 2 1
```

```
DECR 1
```

```
LOADI 3 1
```

```
SUM 2 3
```

```
STOREI 2 1
```

—

```
HALT
```



# Ottimizzazione del codice – funzione optimize

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- 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` finché il codice non può essere ulteriormente ridotto



# Ottimizzazione del codice – funzione optimize

Paradigma  
funzionale

ML

Dichiarazioni e  
scoping in ML

Tipi strutturati in  
ML

Patterns e matching

Liste

Currying

Funzioni di ordine  
superiore

Polimorfismo  
parametrico

Encapsulation e  
interfacce

Eccezioni e  
integrazione con  
type checking

Esempio: un  
semplice compilatore

```
local
 (* definizione singolo passo di ottimizzazione *)
 fun opt1 [] = []
 | opt1 (INCR(x)::STOREI(y,_)::DECR(x1)::cont) =
 let val cont' = opt1 cont in
 if x=x1 andalso x <> y
 then STOREI(y,_)::cont'
 else INCR(x)::STOREI(y,_)::DECR(x1)::cont'
 end
 | opt1 (STOREI(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 ottimizz.*)
 if length(code') = length(code) (* se nessun progresso *)
 then code' (* termina *)
 else optimize code' (* altrimenti riprova *)
 end
end
```



# Efficacia dell'ottimizzazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ecco il risultato per la solita espressione  $3+5*2$ . A sinistra il codice non ottimizzato, a destra quello ottimizzato

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
INCR 1
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
LOADI 3 1
MUL 2 3
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```

Guadagno: 30% di istruzioni in meno



# Combinare le fasi con composizione di funzioni

- 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
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Conclusioni sull'esempio di compilazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare
  - ◆ in Java, che pure è un ottimo linguaggio, 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 sacrificare il controllo di tipi forte
- Per queste ragioni linguaggi come ML vengono utilizzati per la prototipizzazione rapida di compilatori e interpreti

---

# Linguaggi di Programmazione I – Lezione 18

Prof. Piero Bonatti  
<mailto://pab@unina.it>

27 aprile 2020



# Panoramica dell'argomento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

# Paradigma logico



# L'essenza del paradigma logico

**programmi** = insiemi di assiomi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



# L'essenza del paradigma logico

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

**programmi** = insiemi di assiomi

**computazioni** = dimostrazioni costruttive di una formula logica  
data – detta *query* – mediante gli assiomi del programma



# L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

**programmi** = insiemi di assiomi

**computazioni** = dimostrazioni costruttive di una formula logica  
data – detta *query* – mediante gli assiomi del programma

## Esempio

### ■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)



# L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

**programmi** = insiemi di assiomi

**computazioni** = dimostrazioni costruttive di una formula logica  
data – detta *query* – mediante gli assiomi del programma

## Esempio

### ■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

### ■ Query 1

- ◆ Socrate è un uomo?



# L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

**programmi** = insiemi di assiomi

**computazioni** = dimostrazioni costruttive di una formula logica  
data – detta *query* – mediante gli assiomi del programma

## Esempio

### ■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

### ■ Query 1

- ◆ Socrate è un uomo?
- ◆ risposta: yes



# L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

**programmi** = insiemi di assiomi

**computazioni** = dimostrazioni costruttive di una formula logica  
data – detta *query* – mediante gli assiomi del programma

## Esempio

### ■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

### ■ Query 1

- ◆ Socrate è un uomo?
- ◆ risposta: yes

### ■ Query 2

- ◆ Socrate è mortale?



# L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

**programmi** = insiemi di assiomi

**computazioni** = dimostrazioni costruttive di una formula logica data – detta *query* – mediante gli assiomi del programma

## Esempio

### ■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

### ■ Query 1

- ◆ Socrate è un uomo?
- ◆ risposta: yes

### ■ Query 2

- ◆ Socrate è mortale?
- ◆ risposta: yes



# Prolog

Paradigma logico

**Prolog**

Implementazione

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Illustreremo assiomi e query mediante il linguaggio di programmazione logica *Prolog*
- Basandoci sul libro *The Art of Prolog* di Sterling e Shapiro, seconda edizione
  - ◆ reperibile in biblioteca e on-line



# Implementazione e Interazione in Prolog

[Paradigma logico](#)

[Prolog](#)

**Implementazione**

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Analoghe a quelle di ML:

- L'implementazione è mista:



# Implementazione e Interazione in Prolog

[Paradigma logico](#)

[Prolog](#)

**Implementazione**

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Analoghe a quelle di ML:

■ L'implementazione è mista:

- ◆ I programmi Prolog vengono compilati in un bytecode
- ◆ che viene interpretato da una macchina virtuale
- ◆ la *Warren abstract machine* (WAM)



# Implementazione e Interazione in Prolog

[Paradigma logico](#)

[Prolog](#)

**Implementazione**

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Analoghe a quelle di ML:

- L'implementazione è mista:
  - ◆ I programmi Prolog vengono compilati in un bytecode
  - ◆ che viene interpretato da una macchina virtuale
  - ◆ la *Warren abstract machine* (WAM)
  - ◆ si possono anche compilare i programmi in codice stand-alone, direttamente eseguibile
- L'interazione con Prolog è analoga a quella con ML
  - ◆ si inviano query all'interprete e si ottengono le relative risposte



# Implementazione e Interazione in Prolog

Paradigma logico

Prolog

**Implementazione**

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

Analoghe a quelle di ML:

- L'implementazione è mista:
  - ◆ I programmi Prolog vengono compilati in un bytecode
  - ◆ che viene interpretato da una macchina virtuale
  - ◆ la *Warren abstract machine* (WAM)
  - ◆ si possono anche compilare i programmi in codice stand-alone, direttamente eseguibile
- L'interazione con Prolog è analoga a quella con ML
  - ◆ si inviano query all'interprete e si ottengono le relative risposte
  - ◆ oppure, se il programma è stand alone, si interagisce mediante la sua UI (user interface)



# Sistema consigliato per il corso

## ■ SWI Prolog (free)

- ◆ implementa il Prolog standard
- ◆ supporta sia interpretazione che compilazione stand-alone

[Paradigma logico](#)

[Prolog](#)

[Implementazione](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Sistema consigliato per il corso

Paradigma logico

Prolog

**Implementazione**

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

## ■ SWI Prolog (free)

- ◆ implementa il Prolog standard
- ◆ supporta sia interpretazione che compilazione stand-alone
- ◆ invocazione da command line:

```
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)

?-
```

(?- è il prompt dell'interprete)



# Sistema consigliato per il corso

Paradigma logico

Prolog

**Implementazione**

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

## ■ SWI Prolog (free)

- ◆ implementa il Prolog standard
- ◆ supporta sia interpretazione che compilazione stand-alone
- ◆ invocazione da command line:

```
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
?-
```

(?- è il prompt dell'interprete)

## ■ per caricare un proprio programma mioprog.pl

```
?- ['mioprog.pl']. oppure
?- consult('mioprog.pl'). quando si carica la prima volta;
?- reconsult('mioprog.pl'). quando si ricarica dopo una correzione
```



# Costrutti base

## ■ Tre tipi di statement:

1. fatti (facts)
2. regole (rules)
3. queries (detti anche goals)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Costrutti base

## ■ Tre tipi di statement:

1. fatti (facts)
2. regole (rules)
3. queries (detti anche goals)

## ■ Una sola struttura dati:

1. termini logici (logical terms)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Fatti

- Asseriscono una relazione tra oggetti

```
father(abraham, isaac). (Abramo è il padre di Isacco)
```

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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

**Fatti**

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Fatti

- Asseriscono una relazione tra oggetti

```
father(abraham, isaac). (Abramo è il padre di Isacco)
```

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

- Altro esempio: le tabelline

```
per(2,1,2). (due per uno due)
per(2,2,4). (due per due quattro)
per(2,3,6). (due per tre sei)
...
```

- I nomi dei predicati devono iniziare per lettera minuscola

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

**Fatti**

[Queries](#)

[Variabili logiche](#)

[I termini](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Fatti

Paradigma logico

Prolog

Costrutti Prolog

**Fatti**

Queries

Variabili logiche

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- Asseriscono una relazione tra oggetti

`father(abraham, isaac).` (Abramo è il padre di Isacco)

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

- Altro esempio: le tabelline

`per(2,1,2).` (due per uno due)  
`per(2,2,4).` (due per due quattro)  
`per(2,3,6).` (due per tre sei)  
...

- I nomi dei predicati devono iniziare per lettera minuscola
- Gli argomenti `abraham` e `isaac` iniziano con lettera minuscola perché sono *costanti*, come i numeri
  - ◆ introduremo le *variabili* più avanti



## Fatti

- Con i fatti possiamo definire un *database*
  - ◆ l'esempio più semplice di *programma logico*

|                          |                 |
|--------------------------|-----------------|
| father(terach, abraham). | male(terach).   |
| father(terach, nachor).  | male(abraham).  |
| father(terach, haran).   | male(nachor).   |
| father(abraham, isaac).  | male(haran).    |
| father(haran, lot).      | male(isaac).    |
| father(haran, milcah).   | male(lot).      |
| father(haran, yiscah).   |                 |
|                          | female(sarah).  |
| mother(sarah, isaac).    | female(milcah). |
|                          | female(yiscah). |

### Program 1.1 A biblical family database

Ogni predicato corrisponde a una *tabella relazionale*



## Queries

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).
true.
```

```
?- father(isaac,abraham).
false.
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

**Queries**

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

**Queries**

Variabili logiche

I termini

Unificazione

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).
true.
```

```
?- father(isaac,abraham).
false.
```

- Le query hanno la stessa forma dei fatti. Sono entrambi dei cosiddetti *atomi logici*.
  - ◆ nei programmi sono asserzioni (fatti)
  - ◆ nelle query sono domande



# Queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

**Queries**

Variabili logiche

I termini

Unificazione

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).
true.
```

```
?- father(isaac,abraham).
false.
```

- Le query hanno la stessa forma dei fatti. Sono entrambi dei cosiddetti *atomi logici*.
  - ◆ nei programmi sono asserzioni (fatti)
  - ◆ nelle query sono domande
- Nell'esempio qui sopra, l'interprete trova il primo atomo nel programma e non trova il secondo (perciò le risposte diverse)



# Queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

**Queries**

Variabili logiche

I termini

Unificazione

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).
true.
```

```
?- father(isaac,abraham).
false.
```

- Le query hanno la stessa forma dei fatti. Sono entrambi dei cosiddetti *atomi logici*.
  - ◆ nei programmi sono asserzioni (fatti)
  - ◆ nelle query sono domande
- Nell'esempio qui sopra, l'interprete trova il primo atomo nel programma e non trova il secondo (perciò le risposte diverse)
- Nota: nel libro le query sono indicate con un punto interrogativo dopo l'atomo, come in `father(abraham,isaac)?`



# Variabili logiche nelle query

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

**Variabili logiche**

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
  - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

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



# Variabili logiche nelle query

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

**Variabili logiche**

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
  - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

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

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



# Variabili logiche nelle query

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

**Variabili logiche**

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
  - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

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

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



# Variabili logiche nelle query

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

**Variabili logiche**

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
  - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

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

```
?- father(terach,X). /* esiste X tale che father(terach,X) ? */
X=abraham;
X=nachor;
X=haran.
```

- La macchina virtuale 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...



# Variabili logiche in generale

- Differenza tra le variabili logiche e le variabili degli 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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

**Variabili logiche**

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Variabili logiche in generale

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

**Variabili logiche**

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

## ■ Differenza tra le variabili logiche e le variabili degli 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. Un credente potrebbe scrivere in un programma:

```
creato_da(X,dio). /* ogni cosa è creata da Dio */
```

## ■ Differenza tra le variabili nei fatti e nelle query:

- ◆ nelle query: *esistenzialmente quantificate*
  - *esiste* un X tale che ... ?
- ◆ nei fatti: *universalmente quantificate*
  - *per ogni* X vale che ...



# I termini

- I termini sono l'unica struttura dati in Prolog
- Si costruiscono con costanti, variabili (logiche) e *funtori*
  - ◆ ruolo simile ai costruttori di ML
  - ◆ caratterizzati da nome e *arietà* (il numero di argomenti)

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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# I termini

- I termini sono l'unica struttura dati in Prolog
- Si costruiscono con costanti, variabili (logiche) e *funtori*
  - ◆ ruolo simile ai costruttori di ML
  - ◆ caratterizzati da nome e *arietà* (il numero di argomenti)

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

## Esempi

```
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.



# I termini

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october,2000)).
hasColor(object1 , color(rgb,1,0,0)).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# I termini

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

**I termini**

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october,2000)).
hasColor(object1 , color(rgb,1,0,0)).
```

- Quindi la sintassi generale dei fatti è

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

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

(notare il punto dopo il fatto)



# I termini

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

**I termini**

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october ,2000)).
hasColor(object1 , color(rgb ,1 ,0 ,0)).
```

- Quindi la sintassi generale dei fatti è

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

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

(notare il punto dopo il fatto)

- Esempi di query a un programma coi fatti qui sopra

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



# I termini

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

Variabili logiche

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october ,2000)).
hasColor(object1 , color(rgb ,1,0,0)).
```

- Quindi la sintassi generale dei fatti è

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

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

(notare il punto dopo il fatto)

- Esempi di query a un programma coi fatti qui sopra

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

```
?- hasColor(object1 , Y).
Y=color(rgb ,1,0,0)).
```



## I termini

- 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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## I termini

- 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

```
/* fatti */
div(X,X,1).
sum(X,0,X).
sum(0,X,X).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

**I termini**

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## I termini

- 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

```
/* fatti */
div(X,X,1).
sum(X,0,X).
sum(0,X,X).

/* query */
?- hasColor(object1, color(rgb,Y,Y,Y)).
 false. /* nell'unico fatto su object1 c'è 1,0,0 */
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

Programmazione  
nondeterministica

Unicità di Prolog



# I termini

- 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

```
/* fatti */
div(X,X,1).
sum(X,0,X).
sum(0,X,X).

/* query */
?- hasColor(object1, color(rgb,Y,Y,Y)).
 false. /* nell'unico fatto su object1 c'è 1,0,0 */
```

- Differenza tra termini di Prolog e pattern di ML
  - ◆ in ML non posso usare la stessa variabile più volte nello stesso pattern
  - ◆ perchè servono solo a estrarre informazioni da una struttura, non a controllare se elementi diversi sono uguali



# Ground e nonground

- Un termine è *ground* se non contiene variabili
- Altrimenti è *nonground*

```
foo(a,b) /* ground */
bar(X) /* nonground */
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Fatti](#)

[Queries](#)

[Variabili logiche](#)

**I termini**

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



# Ground e nonground

- Un termine è *ground* se non contiene variabili
- Altrimenti è *nonground*

```
foo(a,b) /* ground */
bar(X) /* nonground */
```

- Gli stessi aggettivi e gli stessi criteri si applicano ai fatti, alle query e anche alle regole (che vedremo più avanti)

```
father(abraham,isaac). /* ground */
sum(X,0,X). /* nonground */
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Fatti](#)

[Queries](#)

[Variabili logiche](#)

**I termini**

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



Paradigma logico

Prolog

Costrutti Prolog

**Unificazione**

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

# Unificazione



# Come si risponde alle query

- Per spiegarlo dobbiamo specificare come avviene il matching tra query e fatti e come si costruiscono le risposte
- Dovremo introdurre due nozioni:
  - ◆ sostituzione
  - ◆ unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Sostituzioni

- Una sostituzione è un insieme finito di coppie

$$\theta = \{X_1 = t_1, \dots X_n = t_n\}$$

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  $t_i$ .

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

**Sostituzioni**

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



## Sostituzioni

- Una sostituzione è un insieme finito di coppie

$$\theta = \{X_1 = t_1, \dots X_n = t_n\}$$

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 un 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  $t_i$

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

**Sostituzioni**

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Sostituzioni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

**Sostituzioni**

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Una sostituzione è un insieme finito di coppie

$$\theta = \{X_1 = t_1, \dots X_n = t_n\}$$

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 un 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  $t_i$
- Esempio: se  $\theta = \{x = isaac\}$  e  $E = \text{father}(abraham, x)$  allora

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



## 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)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

**Istanze**

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## 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)
- Esempio:  $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$  e  $E = \text{hasColor(o1, X)}$ .

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

**Istanze**

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## 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)
- Esempio:  $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$  e  $E = \text{hasColor(o1, X)}$ .
  - ◆  $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

**Istanze**

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## 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)
- Esempio:  $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$  e  $E = \text{hasColor(o1, X)}$ .
  - ◆  $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$
  - ◆  $E$  dice che o1 ha un colore non specificato

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

**Istanze**

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## 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)
- Esempio:  $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$  e  $E = \text{hasColor(o1, X)}$ .
  - ◆  $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$
  - ◆  $E$  dice che o1 ha un colore non specificato
  - ◆  $E\theta$  lo specifica parzialmente: è in formato rgb e tutti i 3 valori sono uguali

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

**Istanze**

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Istanze

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

**Istanze**

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- 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)
- Esempio:  $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$  e  $E = \text{hasColor(o1, X)}$ .
  - ◆  $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$
  - ◆  $E$  dice che o1 ha un colore non specificato
  - ◆  $E\theta$  lo specifica parzialmente: è in formato rgb e tutti i 3 valori sono uguali
- Definizione:  $E_1$  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



# L'unificazione

- L'algoritmo di unificazione è quello che effettua il matching
- Prende due espressioni  $E_1$  ed  $E_2$  e – se possibile – restituisce una sostituzione  $\theta$  tale che

$$E_1\theta = E_2\theta$$

detta *unificatore* (unifier)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

[Conjunctive queries](#)

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- L'algoritmo di unificazione è quello che effettua il matching
- Prende due espressioni  $E_1$  ed  $E_2$  e – se possibile – restituisce una sostituzione  $\theta$  tale che

$$E_1\theta = E_2\theta$$

detta *unificatore* (unifier)

- L'unificatore costruito dall'algoritmo viene detto *most general unifier* (mgu) perchè
  - ◆ non sostituisce una variabile con un termine se non è necessario, cioè
  - ◆ vincola il meno possibile il risultato  $E_1\theta$ , lasciando le variabili libere quando può



# L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

[Conjunctive queries](#)

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- L'algoritmo di unificazione è quello che effettua il matching
- Prende due espressioni  $E_1$  ed  $E_2$  e – se possibile – restituisce una sostituzione  $\theta$  tale che

$$E_1\theta = E_2\theta$$

detta *unificatore* (unifier)

- L'unificatore costruito dall'algoritmo viene detto *most general unifier* (mgu) perchè
  - ◆ non sostituisce una variabile con un termine se non è necessario, cioè
  - ◆ vincola il meno possibile il risultato  $E_1\theta$ , lasciando le variabili libere quando può
  - ◆ tecnicamente, ogni altro unificatore  $\theta'$  porta a una *istanza* (cioè caso particolare) di  $E_1\theta$ , cioè
  - ◆ esiste una sostituzione  $\sigma$  tale che  $(E_1\theta)\sigma = E_1\theta'$



# L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## Esempi

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



# L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## Esempi

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



# L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## Esempi

- ◆  $\text{mgu}(\text{sum}(A, B, C), \text{sum}(X, 0, X)) = \{A = X, B = 0, C = X\}$
- ◆  $\text{mgu}(\text{sum}(0, X, 0), \text{sum}(Y, 0, Y)) = \{X = 0, Y = 0\}$
- ◆  $\text{mgu}(\text{sum}(0, X, 0), \text{sum}(Y, Z, 1))$  non esiste a causa del terzo argomento (non si può rendere  $0=1$ )



# Costruzione delle riposte da soli fatti

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

[Istanze](#)

**L'unificazione**

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto  $p(u_1, \dots, u_m)$  con lo stesso funtore (cioè  $p = q$  e  $m = n$ ). Se se ne trova uno:



# Costruzione delle riposte da soli fatti

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto  $p(u_1, \dots, u_m)$  con lo stesso funtore (cioè  $p = q$  e  $m = n$ ). Se se ne trova uno:
2. si calcola  $\theta = \text{mgu}(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$
3. se  $\theta$  esiste, allora:

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog



# Costruzione delle riposte da soli fatti

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto  $p(u_1, \dots, u_m)$  con lo stesso funtore (cioè  $p = q$  e  $m = n$ ). Se se ne trova uno:
  - ◆ si calcola  $\theta = \text{mgu}(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$
  - 3. se  $\theta$  esiste, allora:
    - ◆ si restituisce  $\theta$  come risposta (se è vuota allora Prolog dice *true*)
    - ◆ poi se l'utente non vuole altre soluzioni si termina
4. altrimenti si cerca il prossimo fatto con lo stesso funtore. Se esiste si ripete da 2, altrimenti si termina

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

**L'unificazione**

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog



# Costruzione delle riposte da soli fatti

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto  $p(u_1, \dots, u_m)$  con lo stesso funtore (cioè  $p = q$  e  $m = n$ ). Se se ne trova uno:
    - ◆ si calcola  $\theta = \text{mgu}(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$
    - 3. se  $\theta$  esiste, allora:
      - ◆ si restituisce  $\theta$  come risposta (se è vuota allora Prolog dice *true*)
      - ◆ poi se l'utente non vuole altre soluzioni si termina
  4. altrimenti si cerca il prossimo fatto con lo stesso funtore. Se esiste si ripete da 2, altrimenti si termina
- Prolog dice *false* quando nessun fatto unifica con la query



# Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

**Search tree**

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

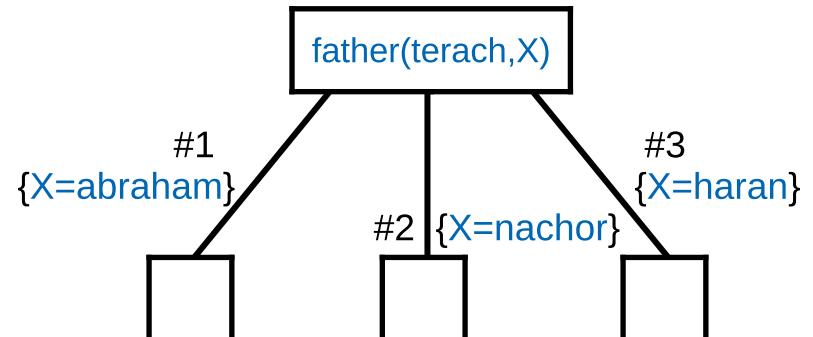
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

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

```
/* programma */
/*1*/ father(terach ,abraham).
/*2*/ father(terach ,nachor).
/*3*/ father(terach ,haran).
/*4*/ father(abraham ,isaac).
 ...
```





# Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

**Search tree**

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

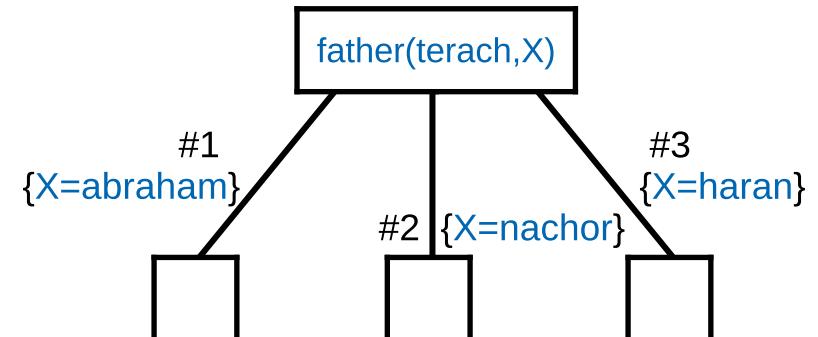
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

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

```
/* programma */
/*1*/ father(terach ,abraham).
/*2*/ father(terach ,nachor).
/*3*/ father(terach ,haran).
/*4*/ father(abraham ,isaac).
 ...
```



- 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



# Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

**Search tree**

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

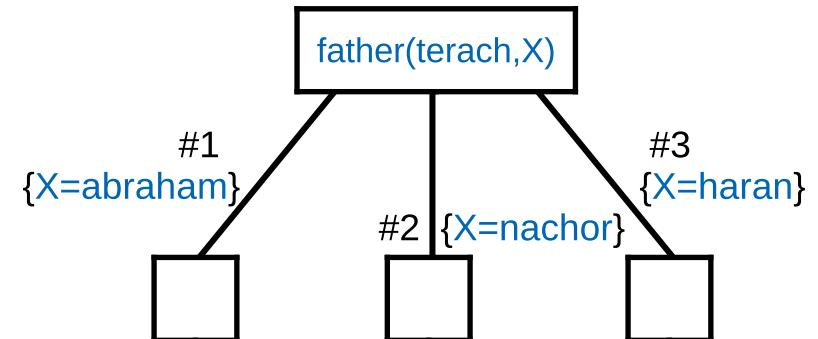
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

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

```
/* programma */
/*1*/ father(terach ,abraham).
/*2*/ father(terach ,nachor).
/*3*/ father(terach ,haran).
/*4*/ father(abraham ,isaac).
 ...
```



- 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



# Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

**Search tree**

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

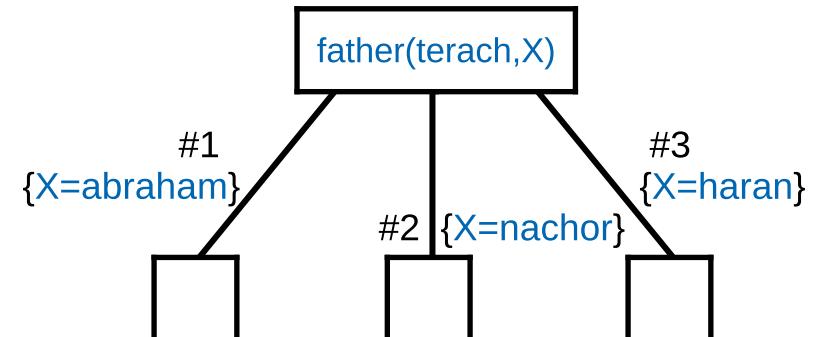
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

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

```
/* programma */
/*1*/ father(terach ,abraham).
/*2*/ father(terach ,nachor).
/*3*/ father(terach ,haran).
/*4*/ father(abraham ,isaac).
 ...
```



- 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
- Le computazioni di Prolog corrispondono a una visita depth-first da sinistra a destra (ogni ramo di successo una risposta)



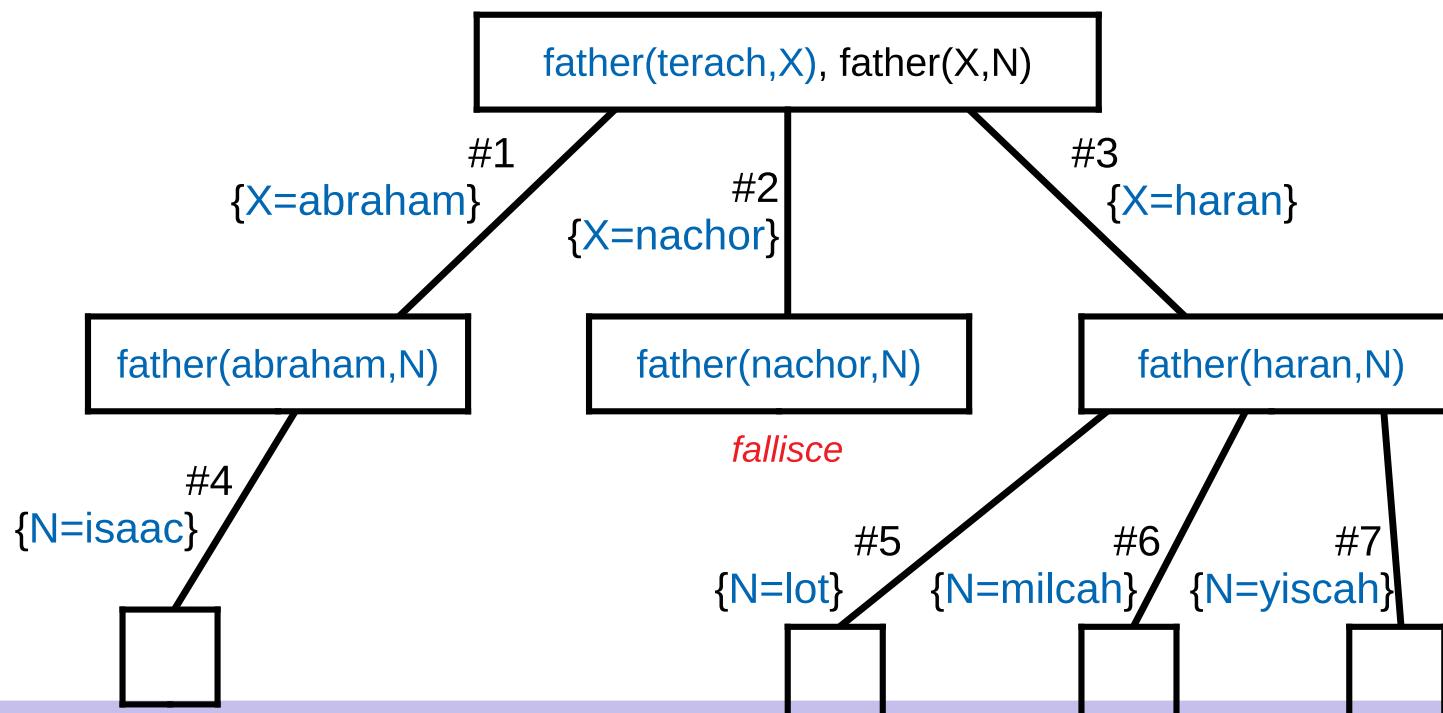
# Conjunctive queries

- Le query possono contenere più formule atomiche (*goals*)
- Esempio: chi sono i nipoti di terach?

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

(la virgola è un *and*) è come un *join*

```
/* programma */
/*1*/ father(terach, abraham).
/*2*/ father(terach, nachor).
/*3*/ father(terach, haran).
/*4*/ father(abraham, isaac).
/*5*/ father(haran, lot).
/*6*/ father(haran, milcah).
/*7*/ father(haran, yiscah).
...
```





# Un'altra visione delle computazioni in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

**Conjunctive queries**

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



# Conjunctive queries

- Le figlie di haran le trovo con la query congiuntiva ... ?

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



# Conjunctive queries

- Le figlie di haran le trovo con la query congiuntiva ... ?

```
father(haran ,Y) , female(Y).
```

I valori di Y mi danno le figlie di haran

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



# Conjunctive queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

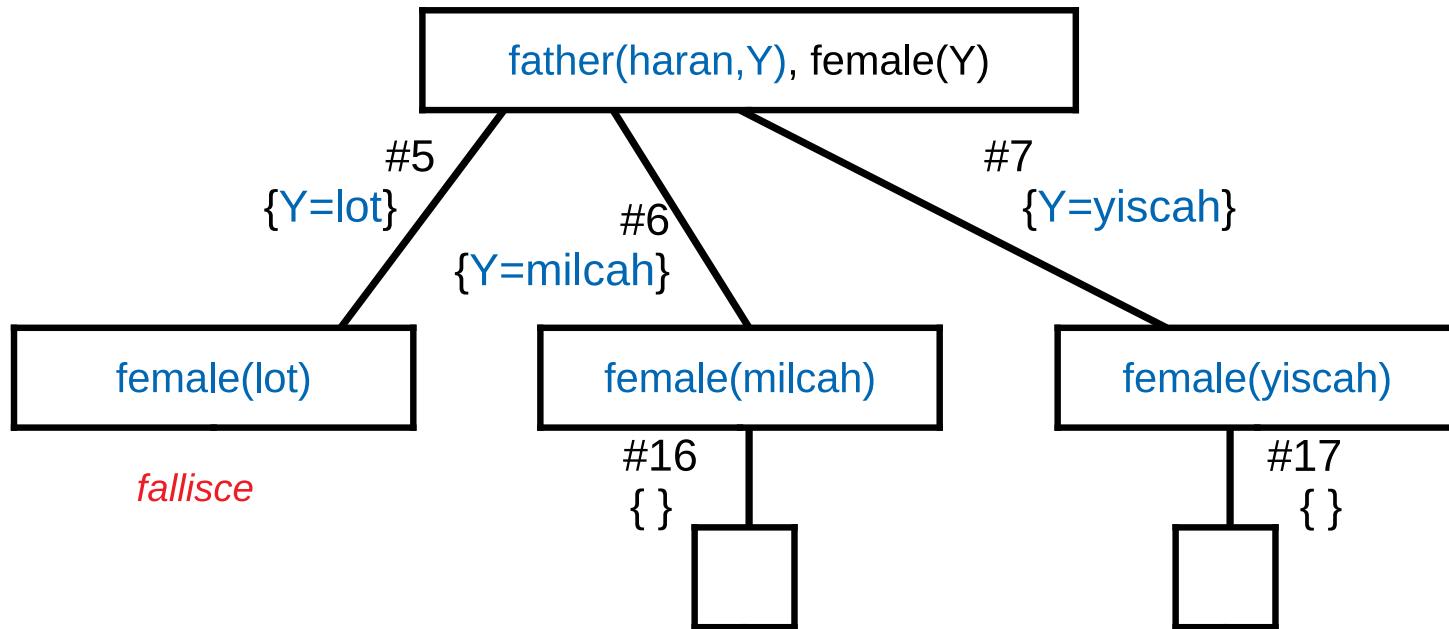
[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Le figlie di haran le trovo con la query congiuntiva ... ?

```
father(haran, Y), female(Y).
```

I valori di Y mi danno le figlie di haran



```
Y = milcah;
Y = yiscah;
false.
```



# Conjunctive queries

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

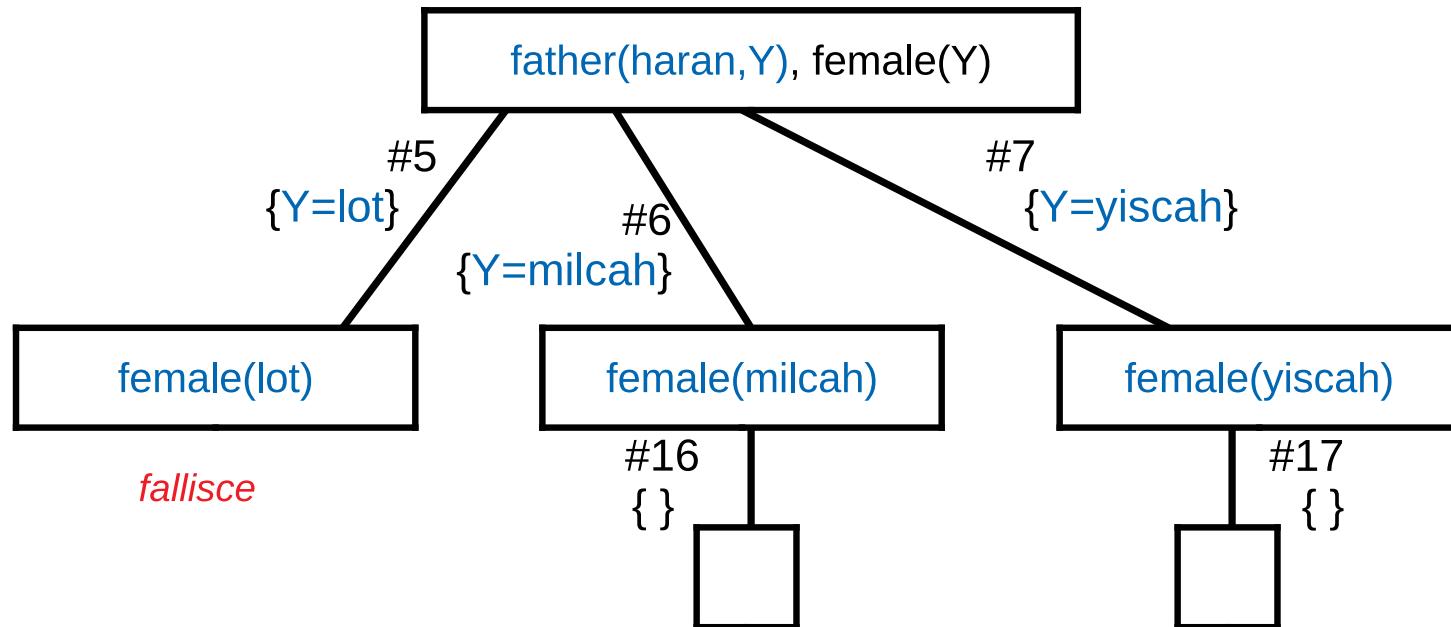
Programmazione nondeterministica

Unicità di Prolog

- Le figlie di haran le trovo con la query congiuntiva ... ?

```
father(haran, Y), female(Y).
```

I valori di Y mi danno le figlie di haran



```
Y = milcah;
Y = yiscah;
false.
```

- Ma... perchè non definire il concetto di figlia nel programma?



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

**Le Regole**

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

# Le Regole



## Le regole

■ *X è figlia di Y se Y è padre di X e X è femmina*

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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



## Le regole

- $X$  è figlia di  $Y$  se  $Y$  è padre di  $X$  e  $X$  è femmina

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

- In generale la sintassi delle regole è

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

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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



## Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

- $X$  è figlia di  $Y$  se  $Y$  è padre di  $X$  e  $X$  è femmina

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

- In generale la sintassi delle regole è

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

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

- ◆ la formula atomica prima di `:-` è detta **testa (head)**



## Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

- $X$  è figlia di  $Y$  se  $Y$  è padre di  $X$  e  $X$  è femmina

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

- In generale la sintassi delle regole è

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

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

- ◆ la formula atomica prima di `:-` è detta **testa (head)**
- ◆ la parte dopo `:-` è detta **corpo (body)**



## Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

- $X$  è figlia di  $Y$  se  $Y$  è padre di  $X$  e  $X$  è femmina

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

- In generale la sintassi delle regole è

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

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

- ◆ la formula atomica prima di :- è detta **testa (head)**
- ◆ la parte dopo :- è detta **corpo (body)**
- ◆ notare che i fatti non sono che regole senza corpo



## Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

- $X$  è figlia di  $Y$  se  $Y$  è padre di  $X$  e  $X$  è femmina

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

- In generale la sintassi delle regole è

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

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

- ◆ la formula atomica prima di :- è detta **testa (head)**
- ◆ la parte dopo :- è detta **corpo (body)**
- ◆ notare che i fatti non sono che regole senza corpo

- **Definizione:** un programma logico è un insieme di regole



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

**Ragionamento**

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

# Ragionamento



# Ragionare con le regole

- Aggiungiamo in coda al programma (posizione 18) la regola

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

Le risposte alla query `daughter(Z, haran)` si trovano così:

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

Programmazione  
nondeterministica

[Unicità di Prolog](#)



# Ragionare con le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Aggiungiamo in coda al programma (posizione 18) la regola

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

Le risposte alla query `daughter(Z, haran)` si trovano così:

```
daughter(Z,haran)
```

Cerca una testa che unifica con la query  
e sostituisce la query col corpo  
istanziato con il mgu



# Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

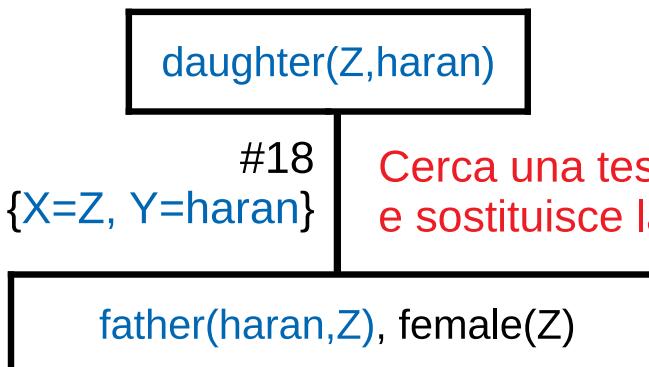
Programmazione nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

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

Le risposte alla query `daughter(Z, haran)` si trovano così:



Cerca una testa che unifica con la query  
e sostituisce la query col corpo  
istanziato con il mgu



# Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

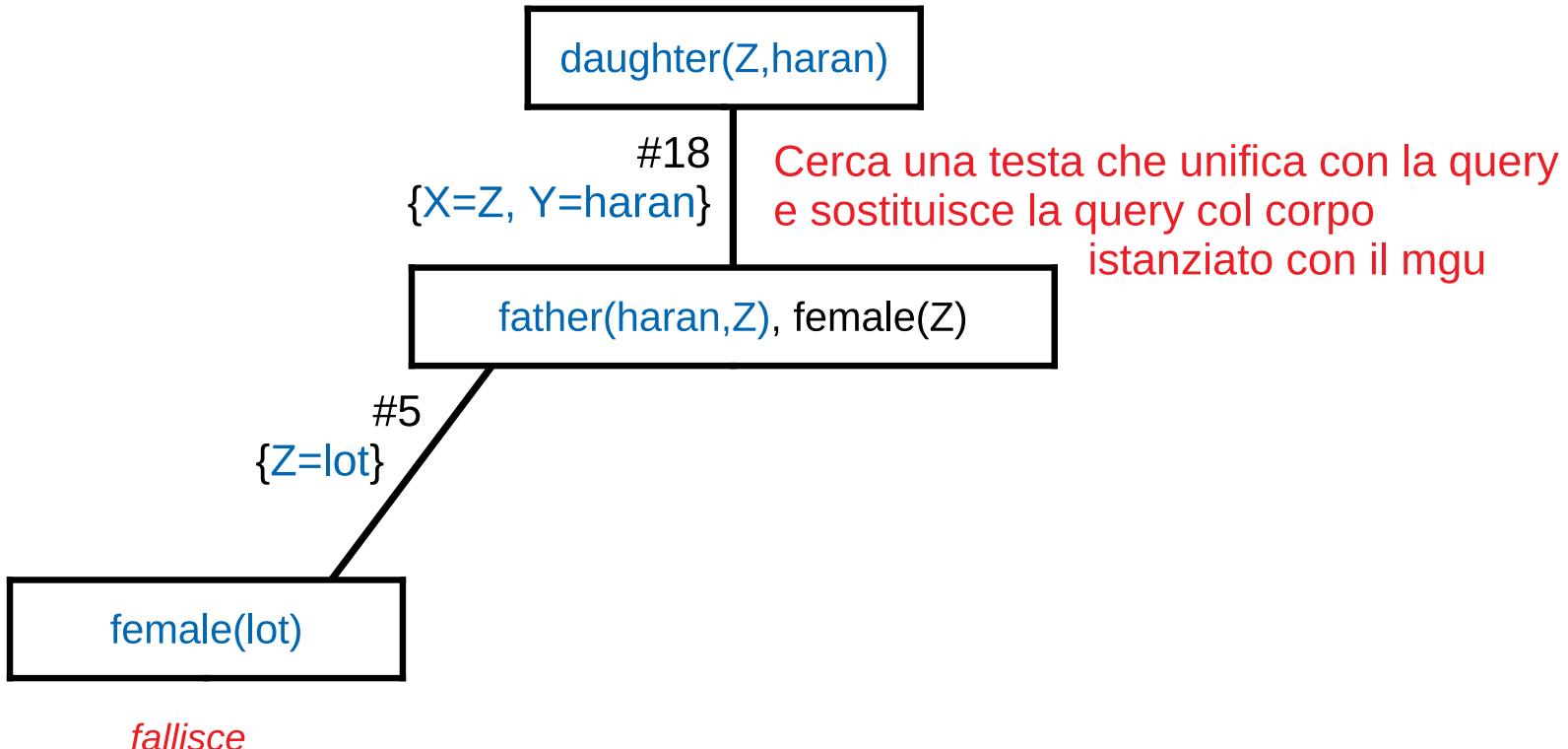
Programmazione nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

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

Le risposte alla query `daughter(Z, haran)` si trovano così:





# Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

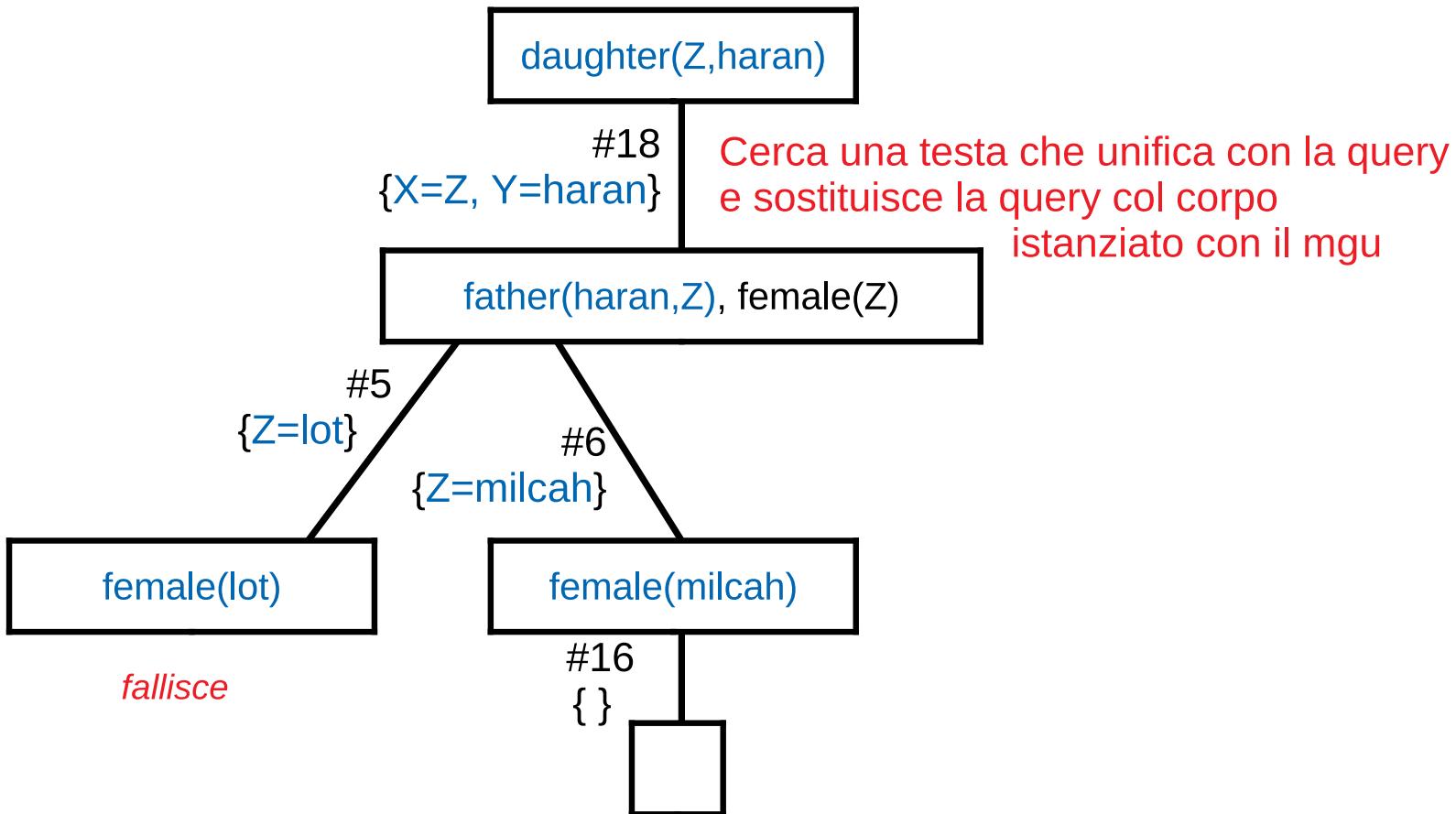
Programmazione  
nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

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

Le risposte alla query `daughter(Z, haran)` si trovano così:





# Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

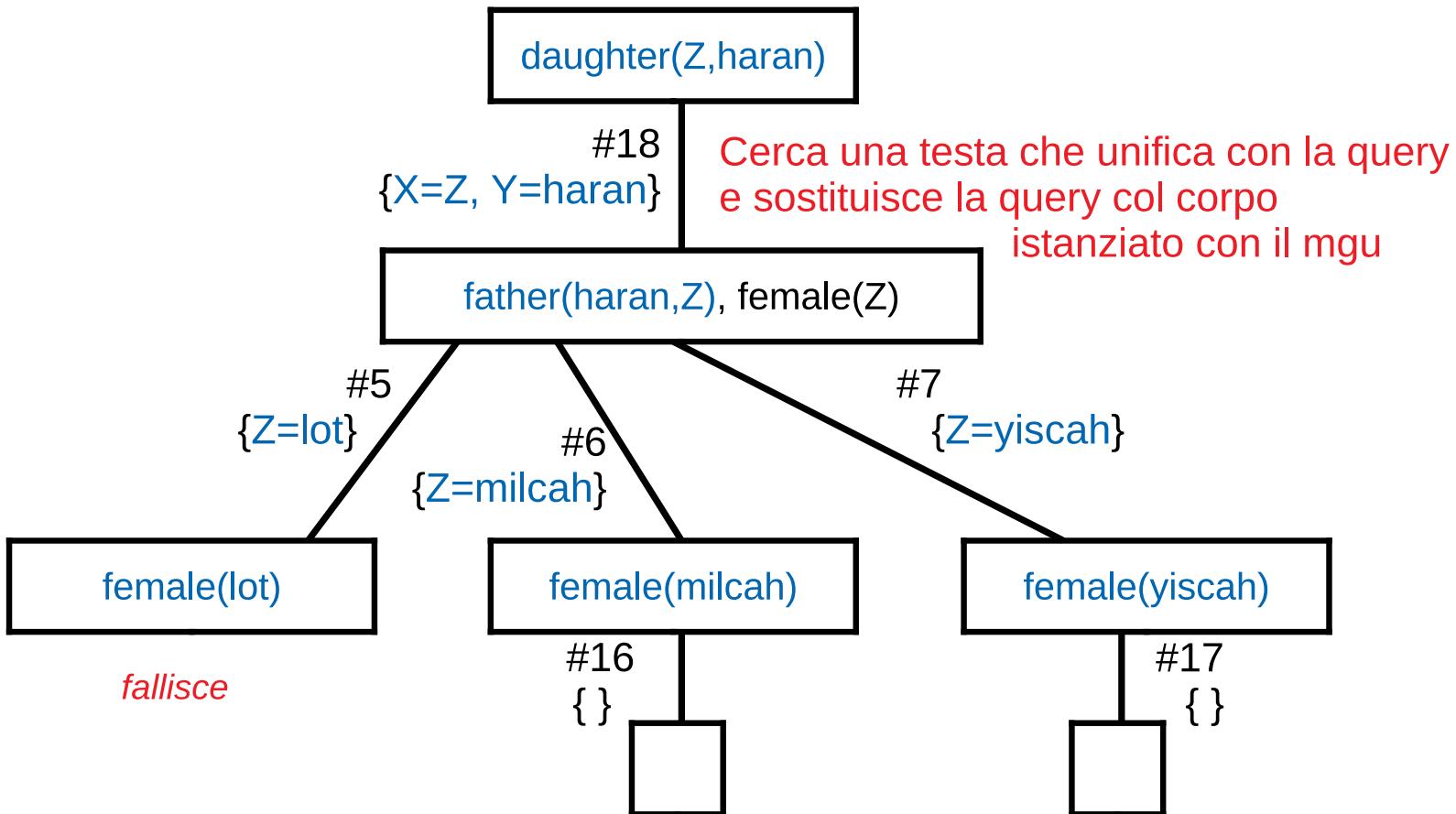
Programmazione  
nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

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

Le risposte alla query `daughter(Z, haran)` si trovano così:





# Ragionare con le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

■ Bisogna fare attenzione con le variabili

◆ quelle della query devono sempre essere diverse da quelle della regola



# Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

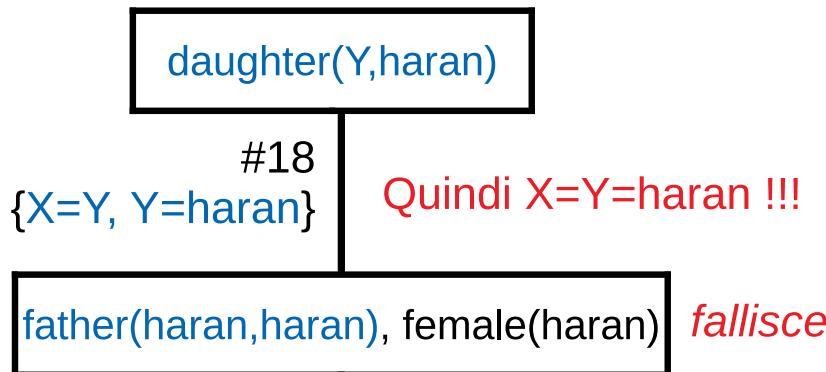
- Bisogna fare attenzione con le variabili

- ◆ quelle della query devono sempre essere diverse da quelle della regola

Esempio di problema se non fosse così:

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

Le risposte alla query `daughter(Y, haran)` verrebbero perse





# Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

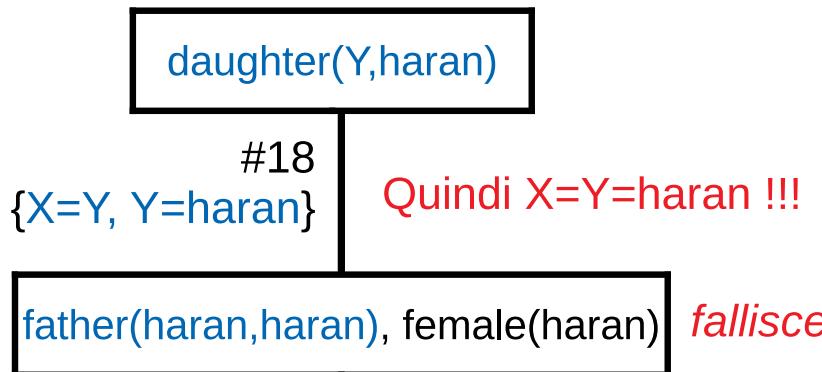
- Bisogna fare attenzione con le variabili

- ◆ quelle della query devono sempre essere diverse da quelle della regola

Esempio di problema se non fosse così:

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

Le risposte alla query `daughter(Y, haran)` verrebbero perse



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

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



# Migliorando l'esempio

- Con la definizione attuale ci perdiamo le figlie delle donne
- **Soluzione 1:** aggiungere un'altra regola in fondo al programma

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

che mi permette di dedurre che X è figlia di Y quando Y è madre di X e X è femmina

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Migliorando l'esempio

- Con la definizione attuale ci perdiamo le figlie delle donne
- **Soluzione 1:** aggiungere un'altra regola in fondo al programma

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

che mi permette di dedurre che X è figlia di Y quando Y è madre di X e X è femmina

- **Soluzione 2:** introdurre il concetto di *genitore* (*parent*)

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

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



# Migliorando l'esempio

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- Con la definizione attuale ci perdiamo le figlie delle donne
- **Soluzione 1:** aggiungere un'altra regola in fondo al programma

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

che mi permette di dedurre che X è figlia di Y quando Y è madre di X e X è femmina

- **Soluzione 2:** introdurre il concetto di *genitore* (parent)

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

/* puo' essere riutilizzato per diverse definizioni */

daughter(X,Y) :- parent(Y,X), female(X).

son(X,Y) :- parent(Y,X), male(X).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```



# Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```



# Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```

Query:

son(isaac,Z)



# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

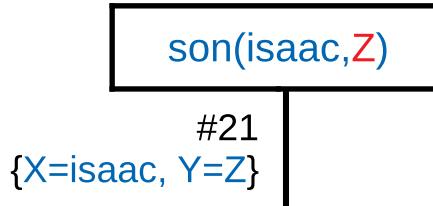
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

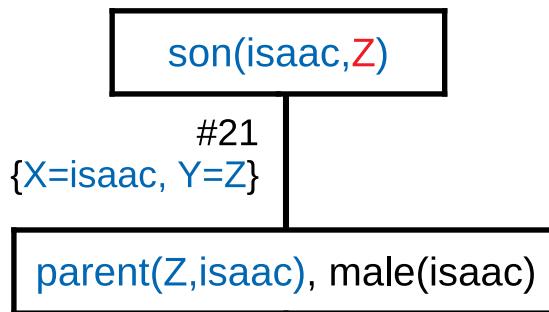
[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

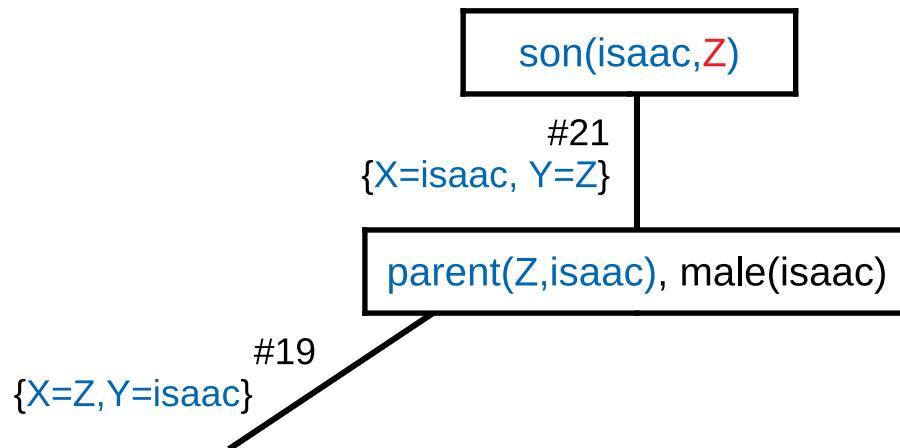
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

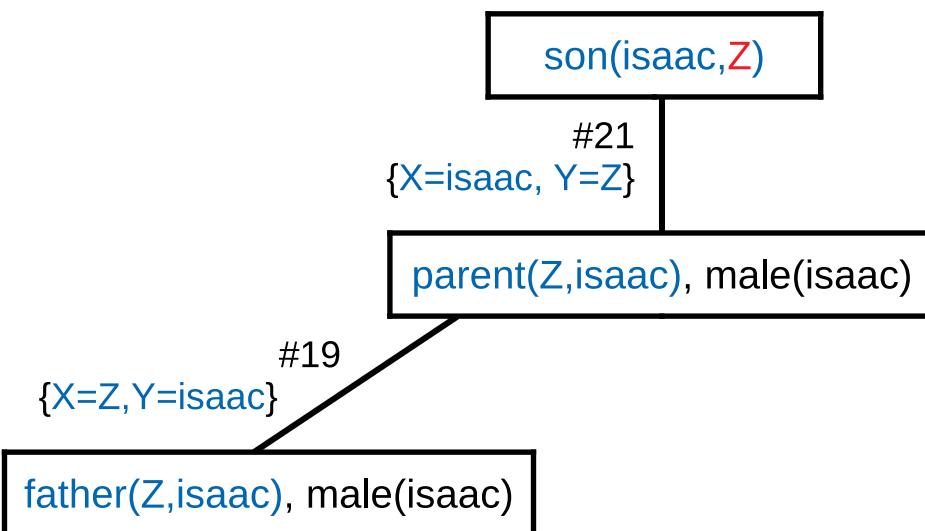
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19 */ parent(X,Y) :- father(X,Y).
/*20 */ parent(X,Y) :- mother(X,Y).
/*21 */ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

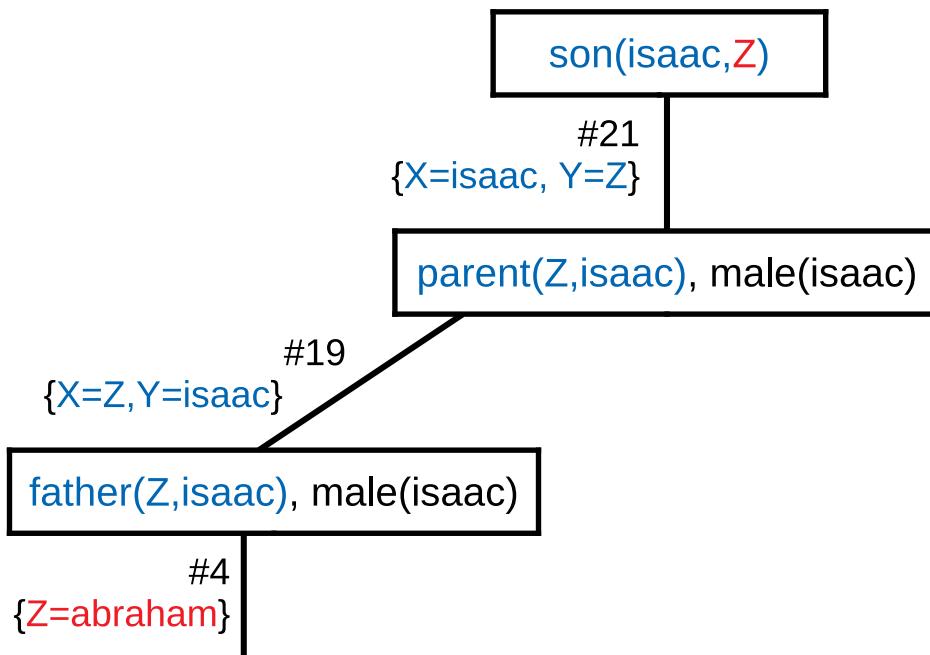
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19 */ parent(X,Y) :- father(X,Y).
/*20 */ parent(X,Y) :- mother(X,Y).
/*21 */ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

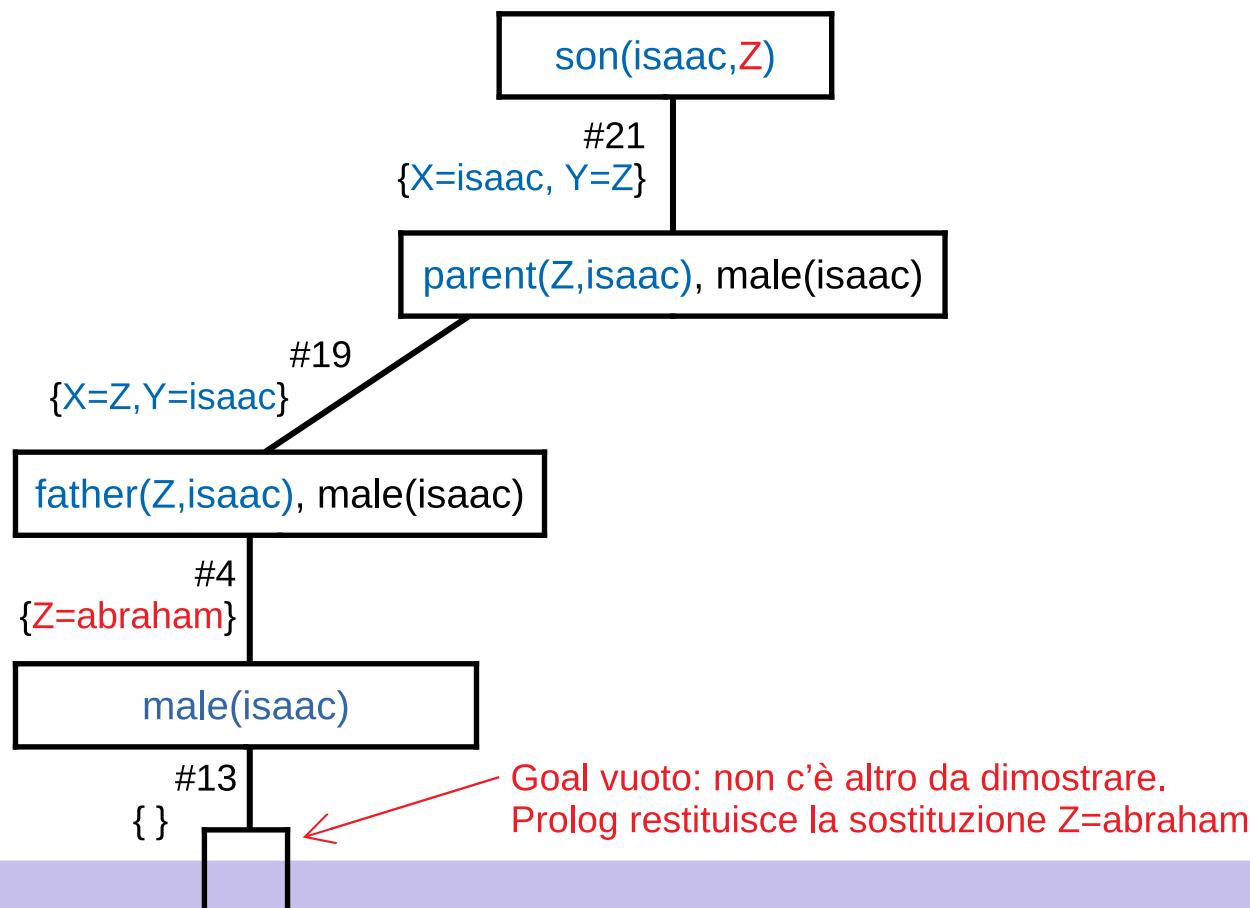
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac). ...
/* 8 */ mother(sarah,isaac). ...
/*13*/ male(isaac). ...
/*19 */ parent(X,Y) :- father(X,Y).
/*20 */ parent(X,Y) :- mother(X,Y).
/*21 */ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

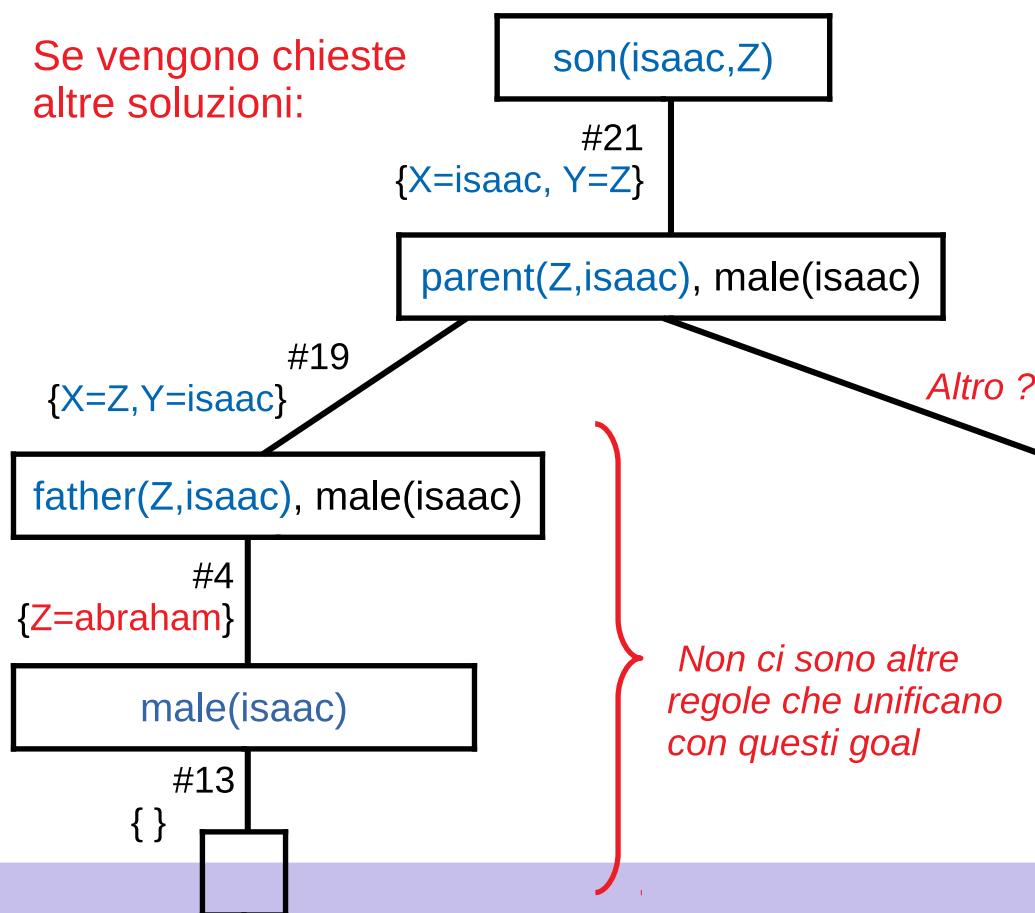
Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19 */ parent(X,Y) :- father(X,Y).
/*20 */ parent(X,Y) :- mother(X,Y).
/*21 */ son(X,Y) :- parent(Y,X), male(X).
```

Se vengono chieste  
altre soluzioni:





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

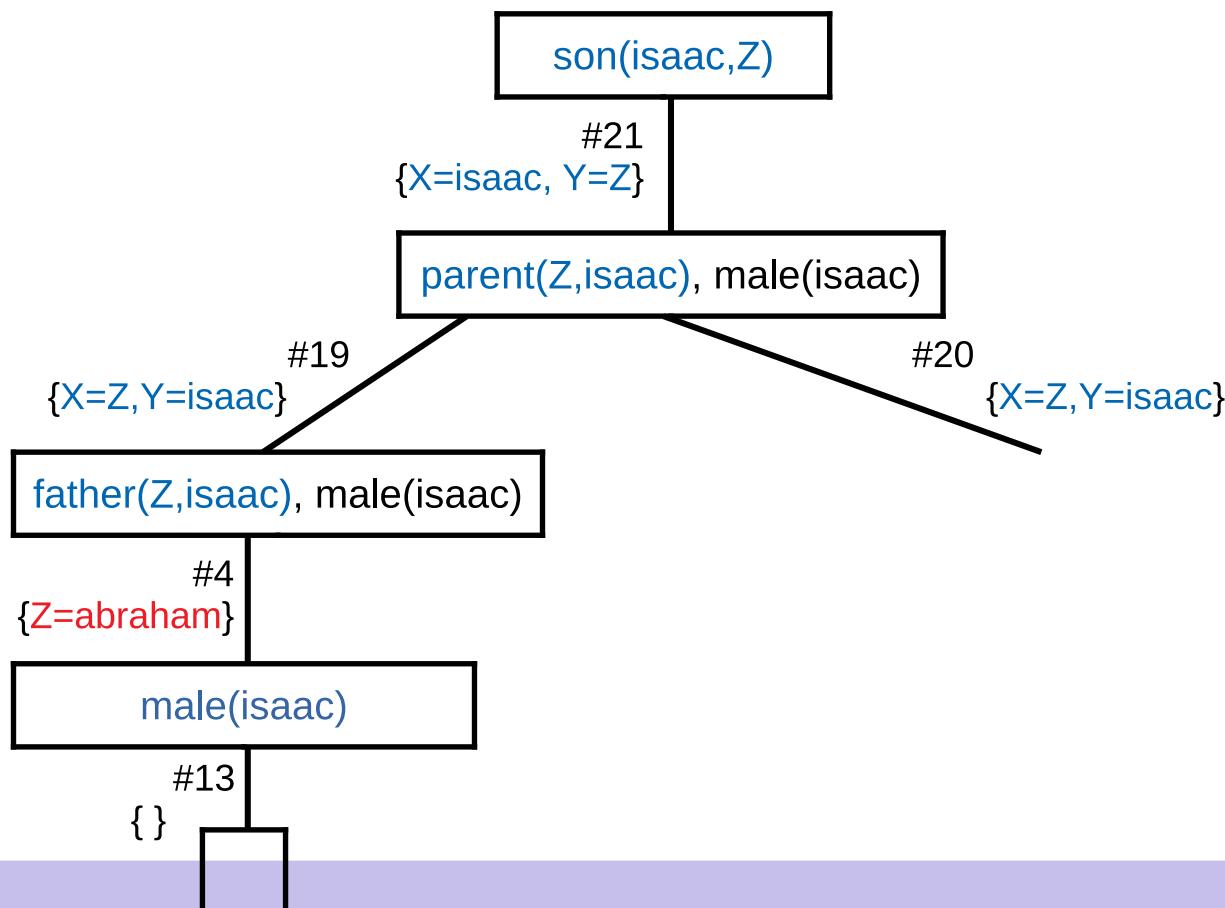
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

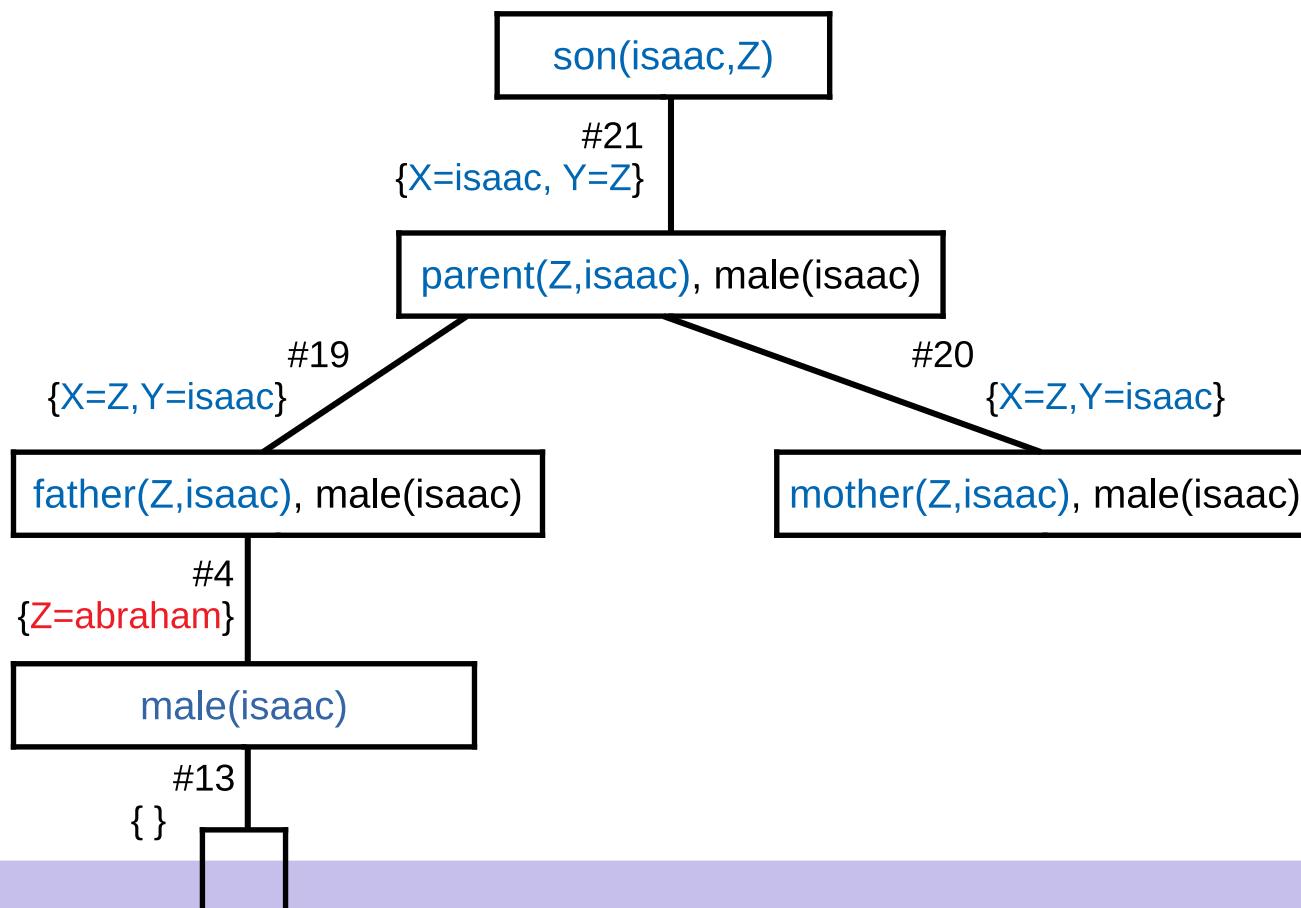
Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

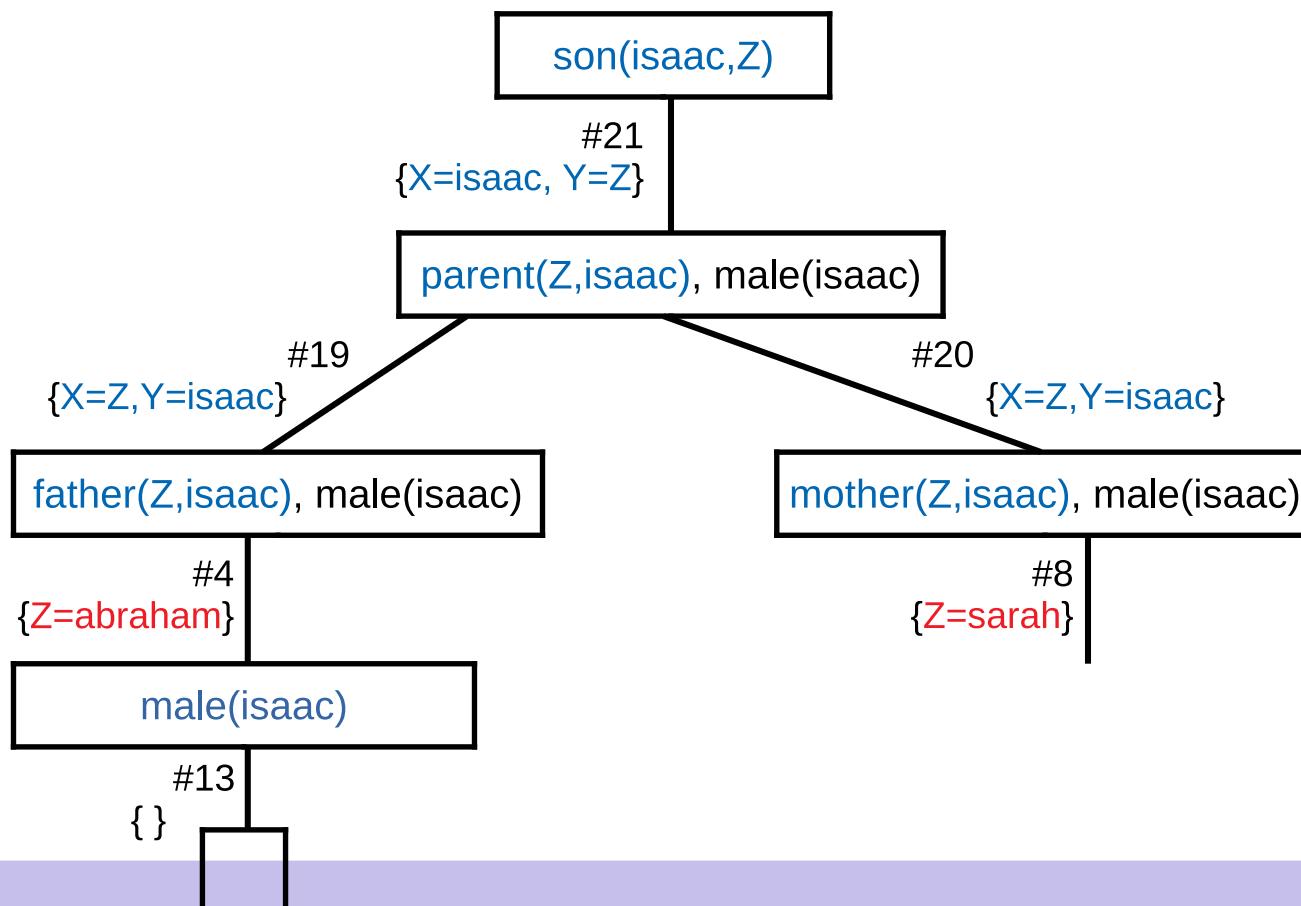
Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

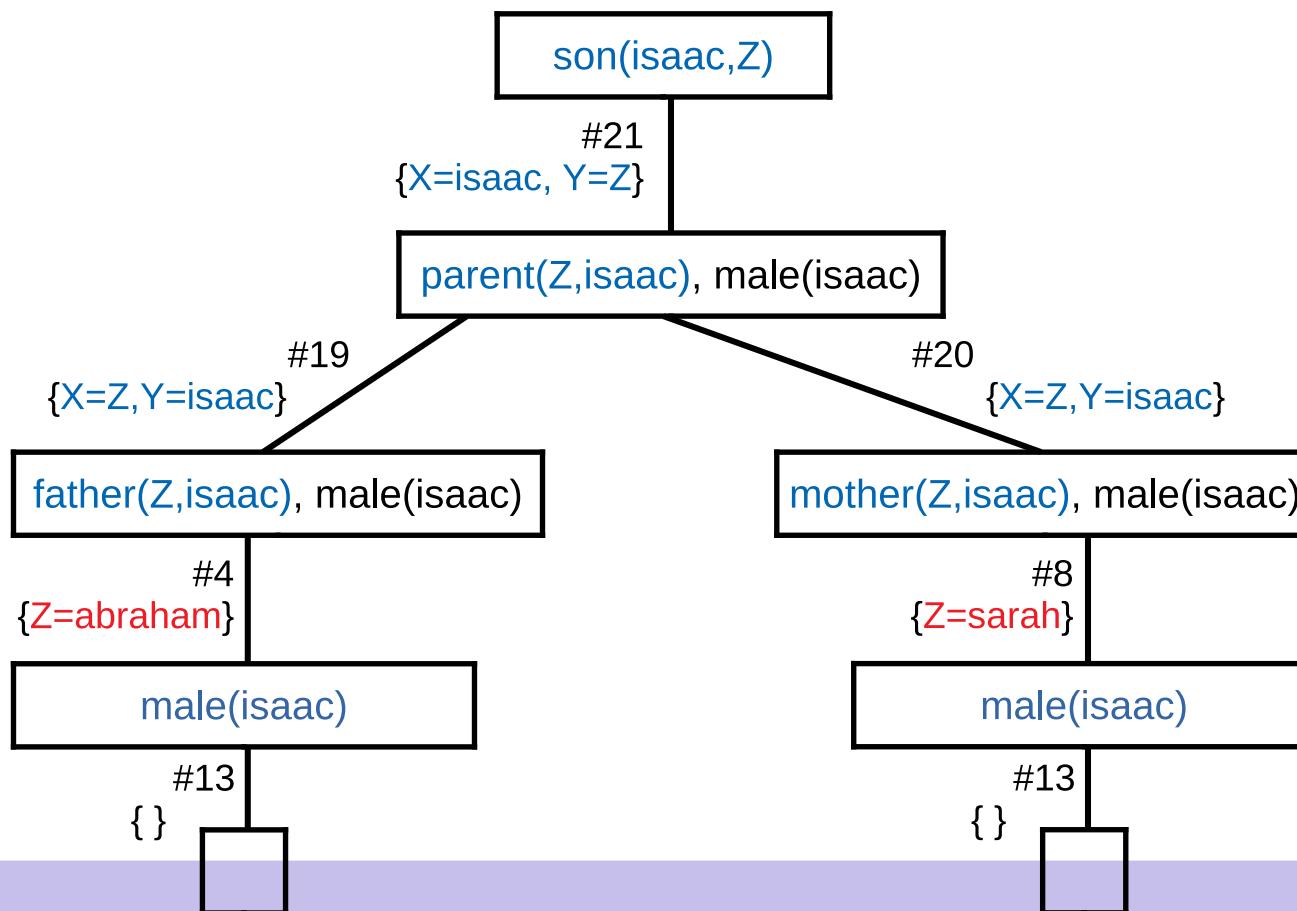
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```





# Esempio di derivazione con standardization apart

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

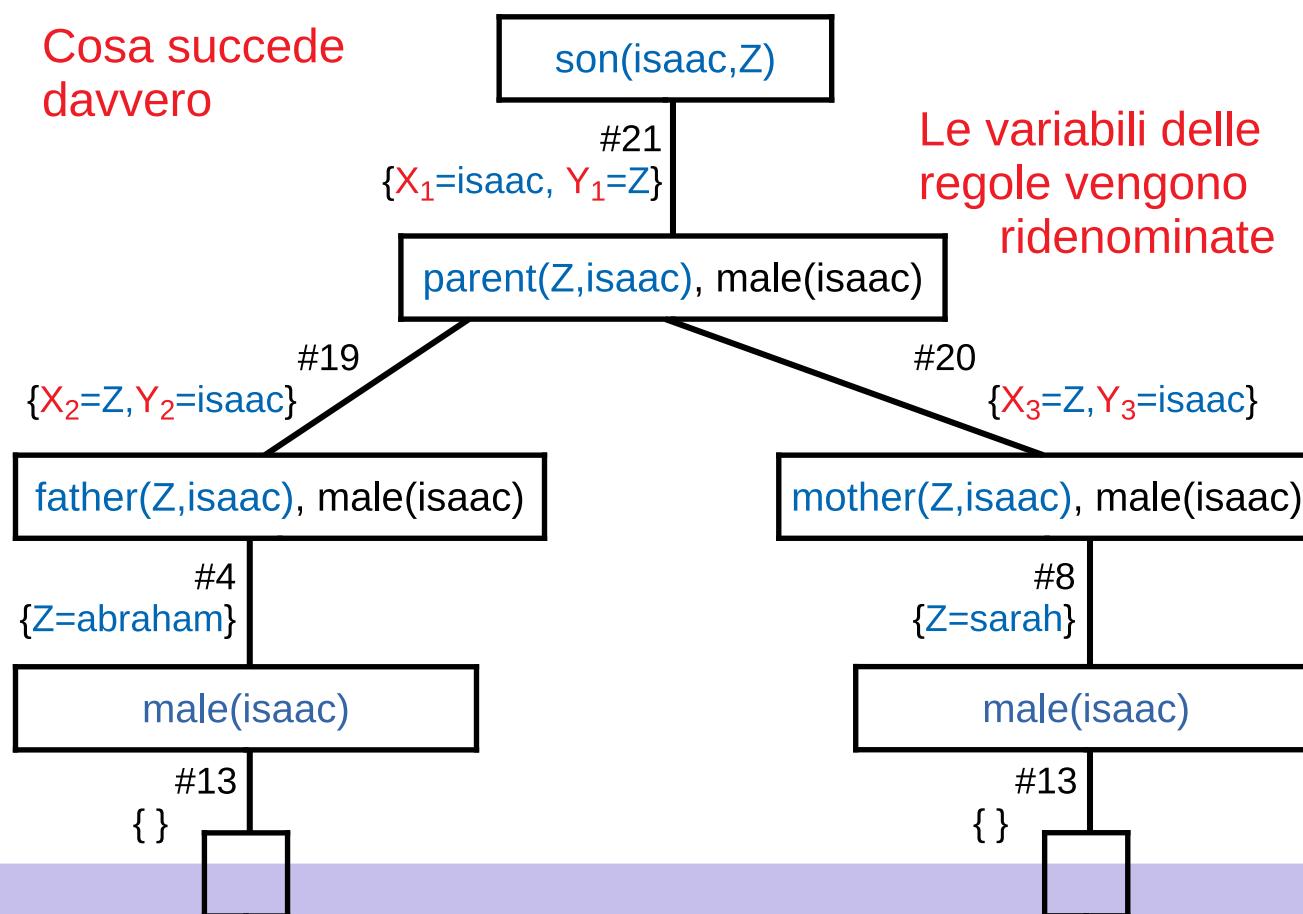
Programmazione  
nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```

Cosa succede  
davvero

Le variabili delle  
regole vengono  
ridenominate





# Un altro modo di vedere le derivazioni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
```

Query: son(isaac,Z)

```
son(X,Y) :- parent(Y,X), male(X).
```

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

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



# Un altro modo di vedere le derivazioni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
```

Query: son(isaac,Z)

call

```
son(X,Y) :- parent(Y,X), male(X).
```

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

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



# Un altro modo di vedere le derivazioni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
```

Query: son(isaac, Z)

call

son(X, Y) :- parent(Y, X), male(X).

{X=isaac}

parent(X, Y) :- father(X, Y).

parent(X, Y) :- mother(X, Y).



# Un altro modo di vedere le derivazioni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
```

Query: son(isaac,Z)

call

son(X,Y) :- parent(Y,X), male(X).

{X=isaac}

1:{Y=abraham}

2:{Y=sarah}

parent(X,Y) :- father(X,Y).

parent(X,Y) :- mother(X,Y).



# Un altro modo di vedere le derivazioni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

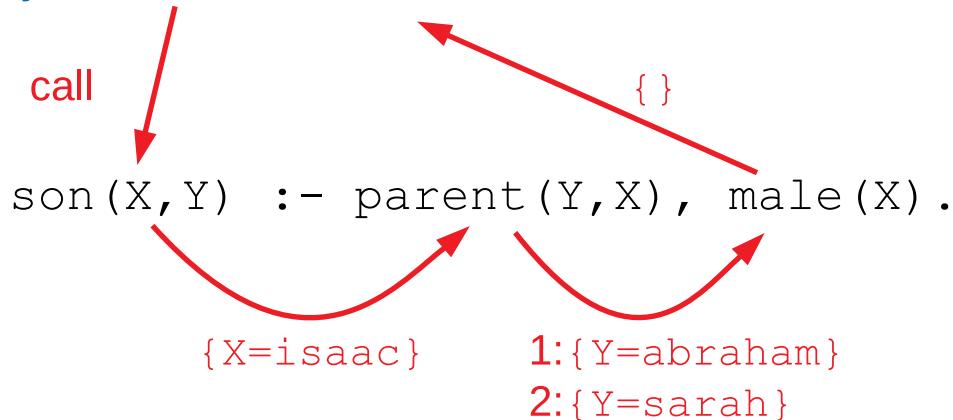
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
```

Query: `son(isaac,Z)`



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

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



# Un altro modo di vedere le derivazioni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

**Derivazioni**

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).
/* 8 */ mother(sarah,isaac).
/*13*/ male(isaac).
```

Query: `son(isaac,Z)`

call

`son(X,Y) :- parent(Y,X), male(X).`

{X=isaac}

Abbiamo dimostrato:

1: `son(isaac,abraham)`  
2: `son(isaac,sarah)`

1:{Y=abraham}  
2:{Y=sarah}

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

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



# Overloading

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Si può usare lo stesso nome di predicato con numeri diversi di argomenti
- Simili prediciati vengono trattati come prediciati diversi
- Esempio: dall'informazione che "X è madre di Y" derivare il concetto di essere madre

```
mother(Mom) :- mother(Mom,X).
/* Mom è una mamma se esiste un X tale che Mom è madre di X */
/* Notare che abbiamo un mother unario e uno binario */
```



# Wildcards

- Notare che nel goal `mother(Mom, X)` il valore di `X` non interessa
- In questi casi possiamo usare wildcards simili a ML

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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

**Wildcards**

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Wildcards

- Notare che nel goal `mother(Mom, X)` il valore di `X` non interessa
- In questi casi possiamo usare wildcards simili a ML

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

- Attenzione: se usiamo due volte una variabile (ad es. `X`) 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 un fatto mother(X, Y) nel database)
```

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

**Wildcards**

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



# Esempio: Analisi di Circuiti

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

**Analisi di circuiti**

Cammini su grafi

Relational algebra

Negazione

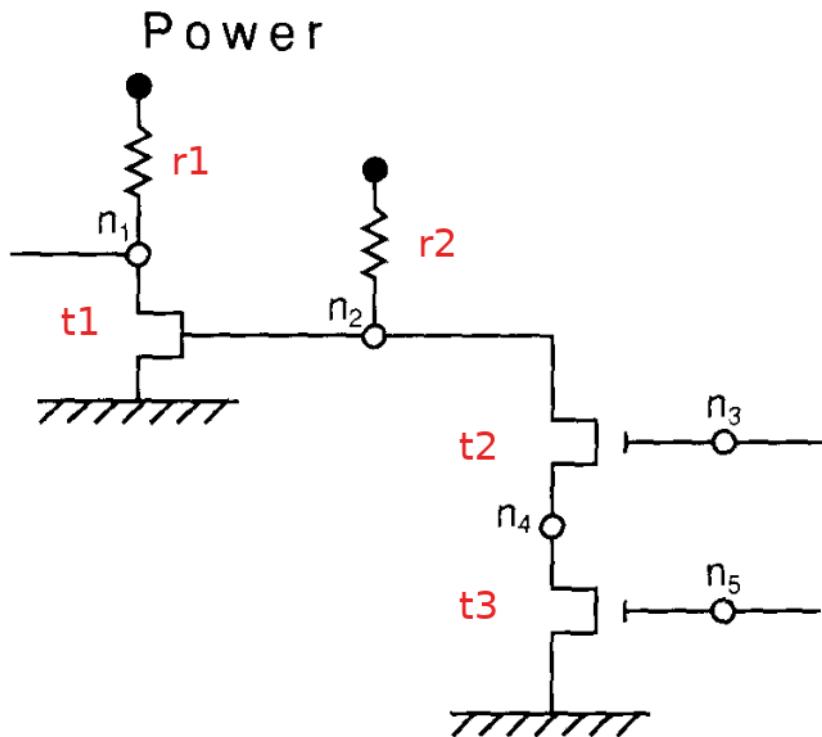
Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

Con i fatti possiamo descrivere circuiti



```
resistor(r1, power, n1).
resistor(r2, power, n2).
transistor(t1, n2, ground, n1).
transistor(t2, n3, n4, n2).
transistor(t3, n5, ground, n4).
```

**Figure 2.2** A logical circuit



# Esempio: Analisi di Circuiti

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

**Analisi di circuiti**

Cammini su grafi

Relational algebra

Negazione

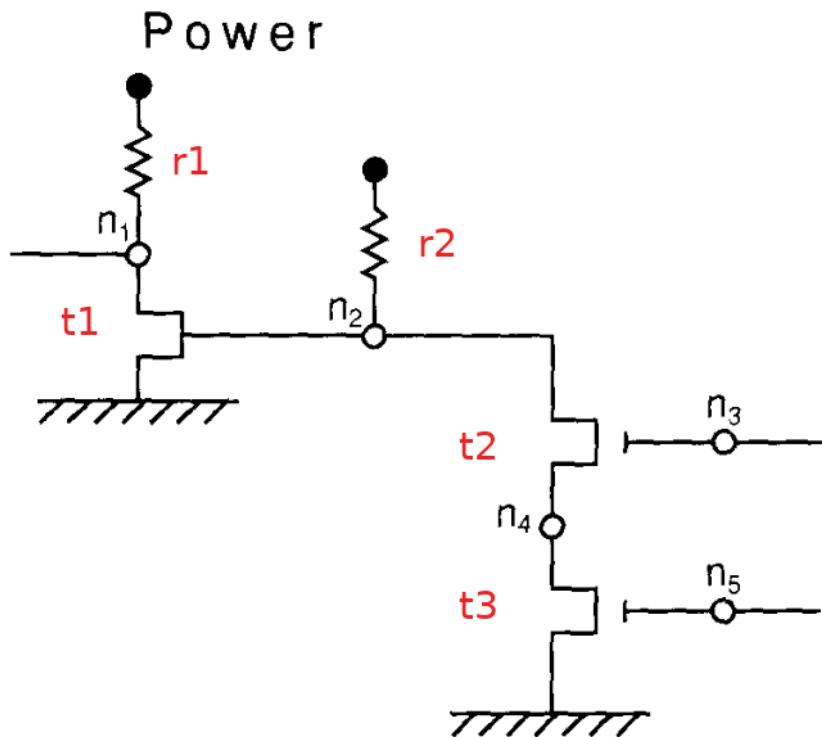
Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

Con i fatti possiamo descrivere circuiti



```
resistor(r1, power, n1).
resistor(r2, power, n2).
transistor(t1, n2, ground, n1).
transistor(t2, n3, n4, n2).
transistor(t3, n5, ground, n4).
```

**Figure 2.2** A logical circuit

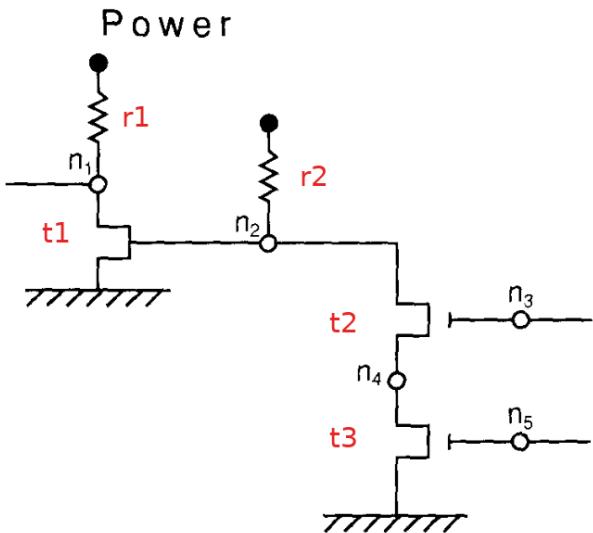
Da questa descrizione fisica vogliamo derivarne una funzionale

- ovvero: cosa fanno questi componenti?



# Esempio: Analisi di Circuiti

## ■ Descrizione funzionale dei circuiti



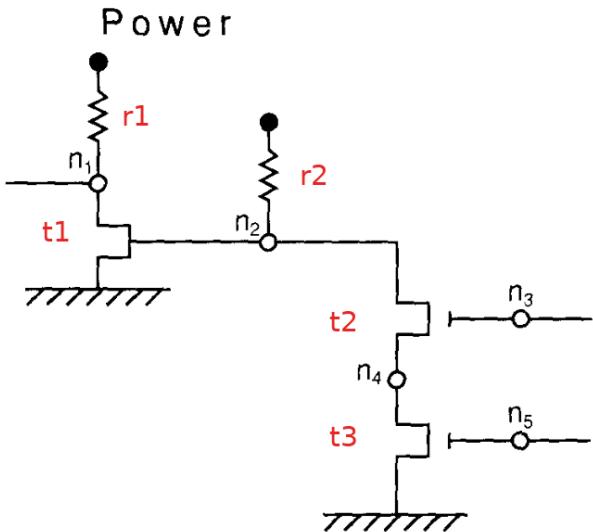
```
inverter(inv(T,R), In, Out) :-
 transistor(T, In, ground, Out),
 resistor(R, power, Out).
```

Figure 2.2 A logical circuit



# Esempio: Analisi di Circuiti

## ■ Descrizione funzionale dei circuiti



```
inverter(inv(T,R), In, Out) :-
 transistor(T, In, ground, Out),
 resistor(R, power, Out).

nand_gate(nand(T1,T2,R), In1, In2, Out) :-
 transistor(T1, In1, X, Out),
 transistor(T2, In2, ground, X),
 resistor(R, power, Out).
```

Figure 2.2 A logical circuit



# Esempio: Analisi di Circuiti

## ■ Descrizione funzionale dei circuiti

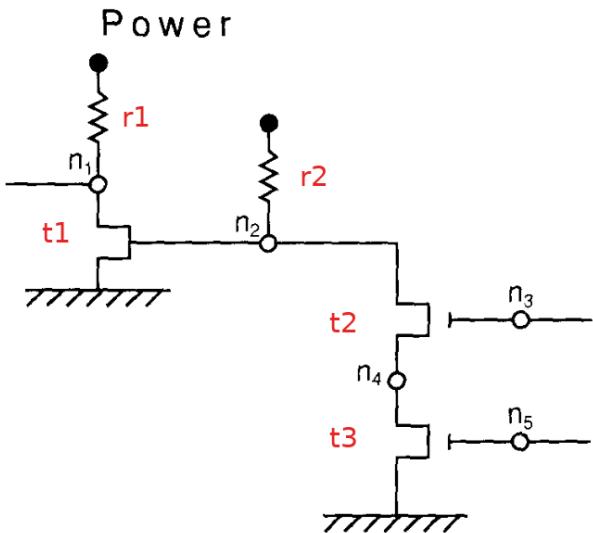


Figure 2.2 A logical circuit

```
inverter(inv(T,R), In, Out) :-
 transistor(T, In, ground, Out),
 resistor(R, power, Out).

nand_gate(nand(T1,T2,R), In1, In2, Out) :-
 transistor(T1, In1, X, Out),
 transistor(T2, In2, ground, X),
 resistor(R, power, Out).

and_gate(and(N,I), In1, In2, Out) :-
 nand_gate(N, In1, In2, X),
 inverter(I,X,Out).
```



# Esempio: Analisi di Circuiti

## ■ Descrizione funzionale dei circuiti

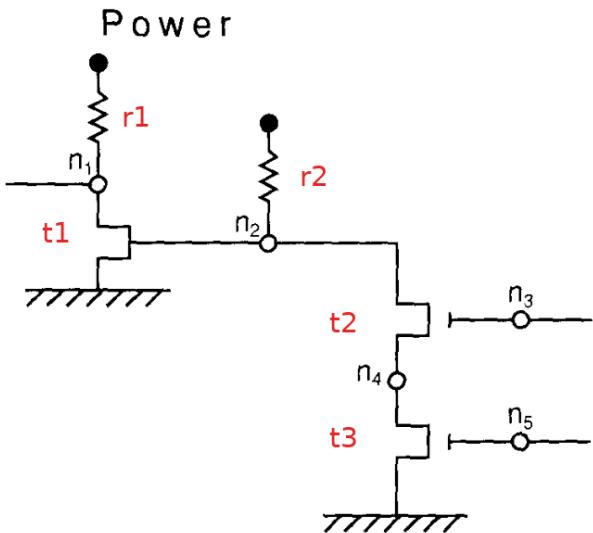


Figure 2.2 A logical circuit

```
inverter(inv(T,R), In, Out) :-
 transistor(T, In, ground, Out),
 resistor(R, power, Out).

nand_gate(nand(T1,T2,R), In1, In2, Out) :-
 transistor(T1, In1, X, Out),
 transistor(T2, In2, ground, X),
 resistor(R,power,Out).

and_gate(and(N,I), In1, In2, Out) :-
 nand_gate(N, In1, In2, X),
 inverter(I,X,Out).
```

Ora possiamo trovare i componenti logici implementati dal circuito

```
?- inverter(X,Y,Z).
X = inv(t1, r1),
Y = n2,
Z = n1.
```



# Esempio: Analisi di Circuiti

## ■ Descrizione funzionale dei circuiti

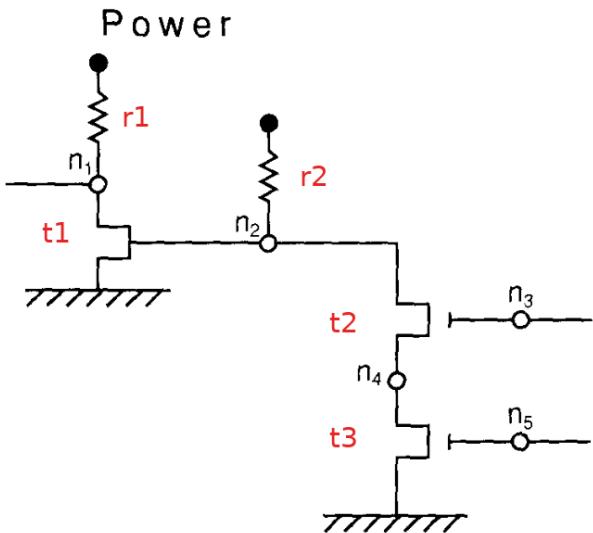


Figure 2.2 A logical circuit

```
inverter(inv(T,R), In, Out) :-
 transistor(T, In, ground, Out),
 resistor(R, power, Out).

nand_gate(nand(T1,T2,R), In1, In2, Out) :-
 transistor(T1, In1, X, Out),
 transistor(T2, In2, ground, X),
 resistor(R,power,Out).

and_gate(and(N,I), In1, In2, Out) :-
 nand_gate(N, In1, In2, X),
 inverter(I,X,Out).
```

Ora possiamo trovare i componenti logici implementati dal circuito

```
?- inverter(X,Y,Z).
X = inv(t1, r1),
Y = n2,
Z = n1.
```

```
?- nand_gate(X,Y,Z,0).
X = nand(t2, t3, r2),
Y = n3,
Z = n5,
0 = n2.
```



# Esempio: Analisi di Circuiti

## ■ Descrizione funzionale dei circuiti

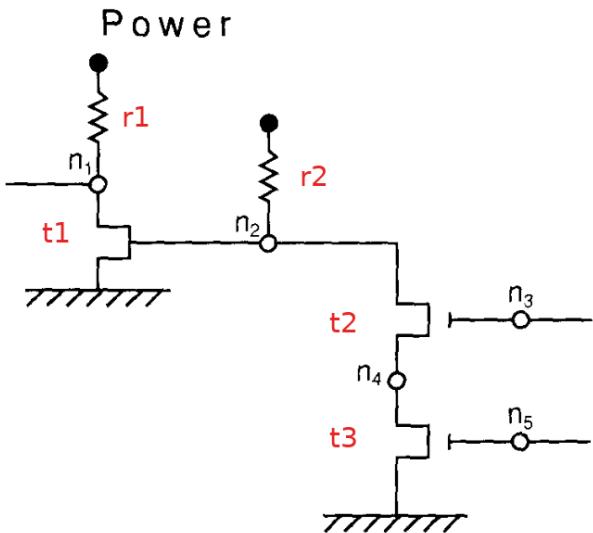


Figure 2.2 A logical circuit

```
inverter(inv(T,R), In, Out) :-
 transistor(T, In, ground, Out),
 resistor(R, power, Out).

nand_gate(nand(T1,T2,R), In1, In2, Out) :-
 transistor(T1, In1, X, Out),
 transistor(T2, In2, ground, X),
 resistor(R,power,Out).

and_gate(and(N,I), In1, In2, Out) :-
 nand_gate(N, In1, In2, X),
 inverter(I,X,Out).
```

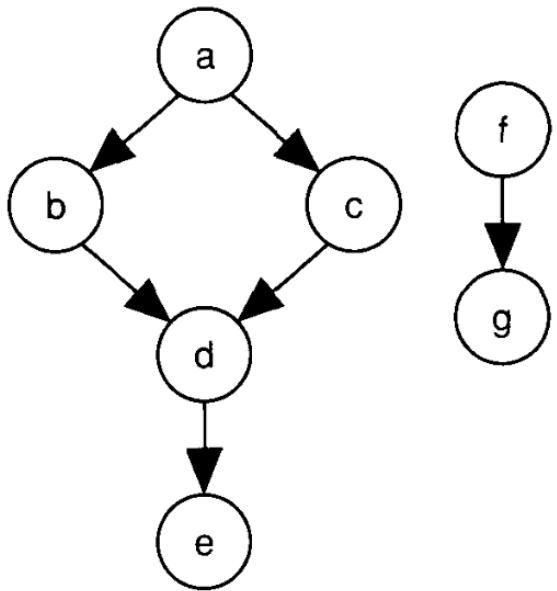
Ora possiamo trovare i componenti logici implementati dal circuito

|                                                                             |                                                                                                 |                                                                                                                  |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <pre>?- inverter(X,Y,Z).<br/>X = inv(t1, r1),<br/>Y = n2,<br/>Z = n1.</pre> | <pre>?- nand_gate(X,Y,Z,0).<br/>X = nand(t2, t3, r2),<br/>Y = n3,<br/>Z = n5,<br/>0 = n2.</pre> | <pre>?- and_gate(X,Y,Z,0).<br/>X =<br/>and(nand(t2,t3,r2),inv(t1,r1)),<br/>Y = n3,<br/>Z = n5,<br/>0 = n1.</pre> |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|



## Regole ricorsive

Cerchiamo ora i cammini in un grafo



```
/* programma che descrive il grafo */

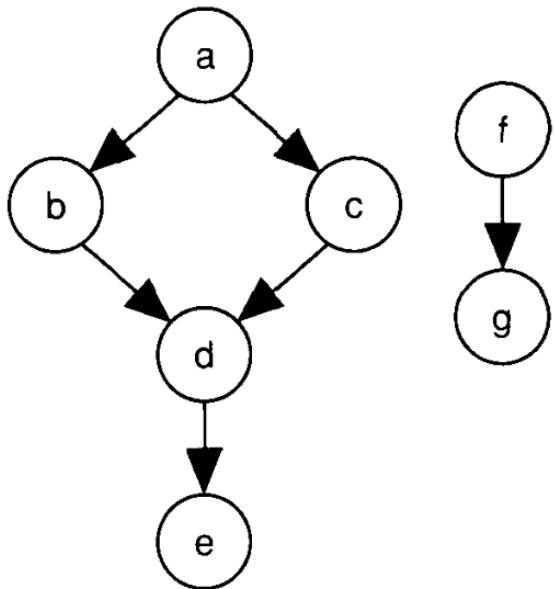
arco(a,b). arco(b,d). arco(d,e).
arco(a,c). arco(c,d). arco(f,g).
```

Figure 2.4 A simple graph



## Regole ricorsive

Cerchiamo ora i cammini in un grafo



```
/* programma che descrive il grafo */

arco(a,b). arco(b,d). arco(d,e).
arco(a,c). arco(c,d). arco(f,g).
```

Verificare se due nodi sono connessi:

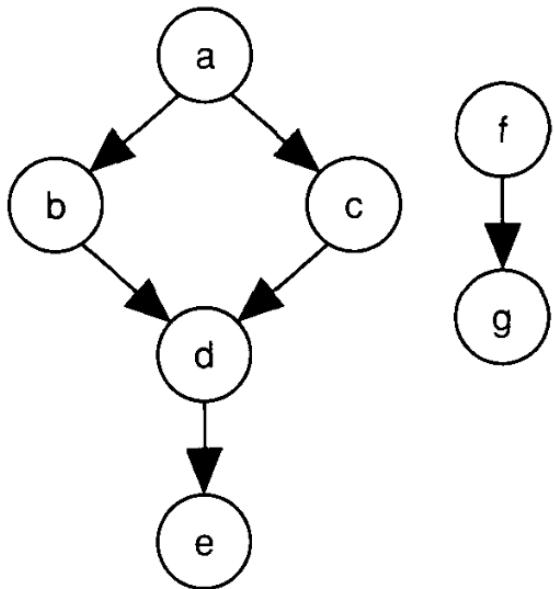
```
connessi(X,X).
```

Figure 2.4 A simple graph



## Regole ricorsive

Cerchiamo ora i cammini in un grafo



```
/* programma che descrive il grafo */

arco(a,b). arco(b,d). arco(d,e).
arco(a,c). arco(c,d). arco(f,g).
```

Verificare se due nodi sono connessi:

```
connessi(X,X).
connessi(X,Y) :- arco(X,Z), connessi(Z,Y).
```

Figure 2.4 A simple graph



## Search tree parziale per la query `connessi(a,d)`

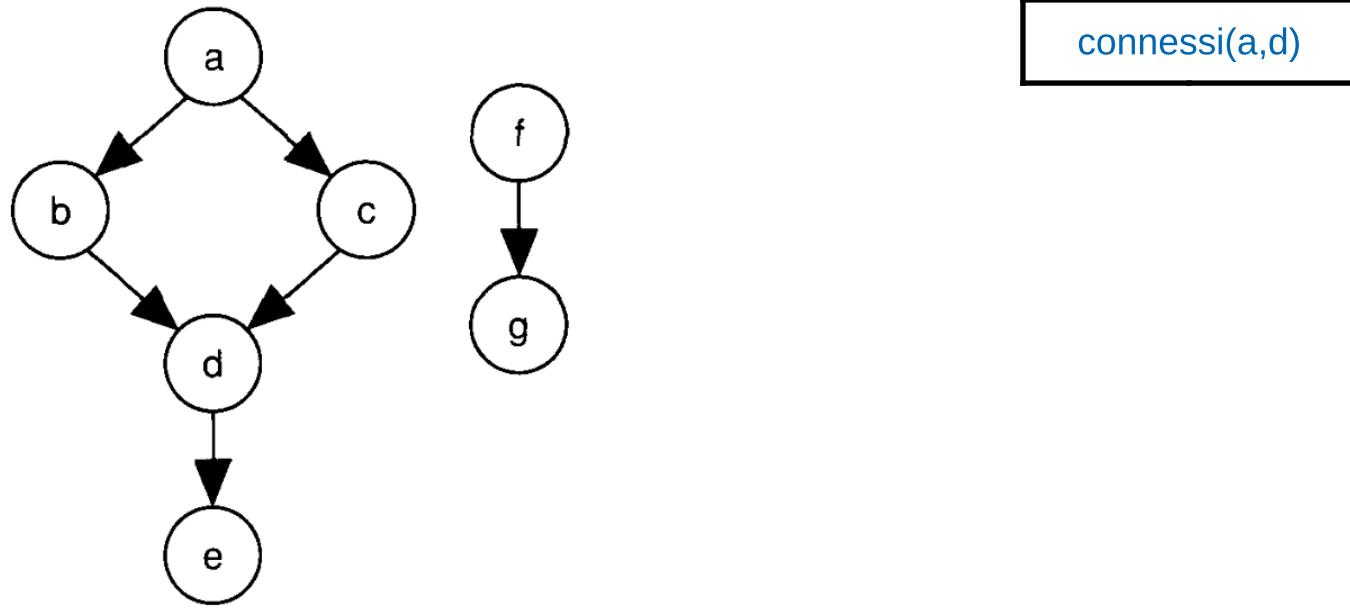


Figure 2.4 A simple graph



## Search tree parziale per la query $\text{connessi}(a,d)$

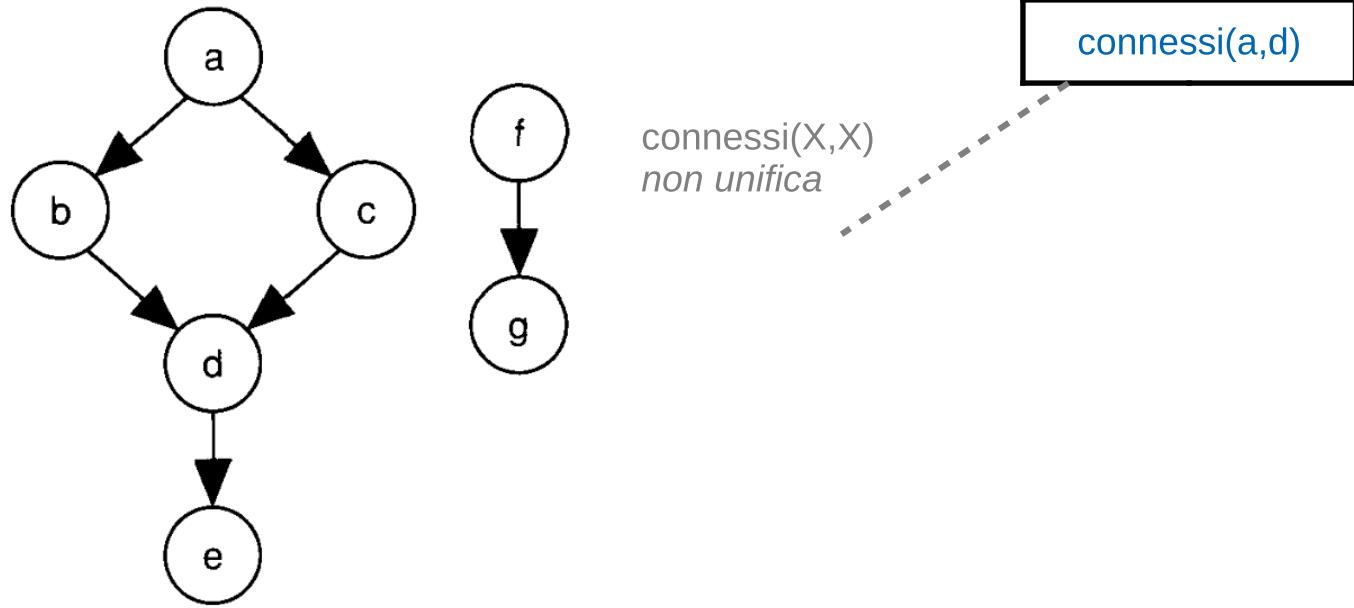
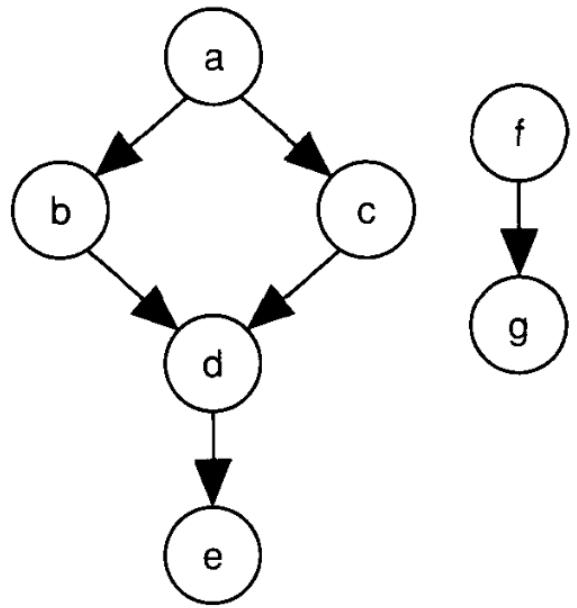


Figure 2.4 A simple graph



## Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`  
non unifica

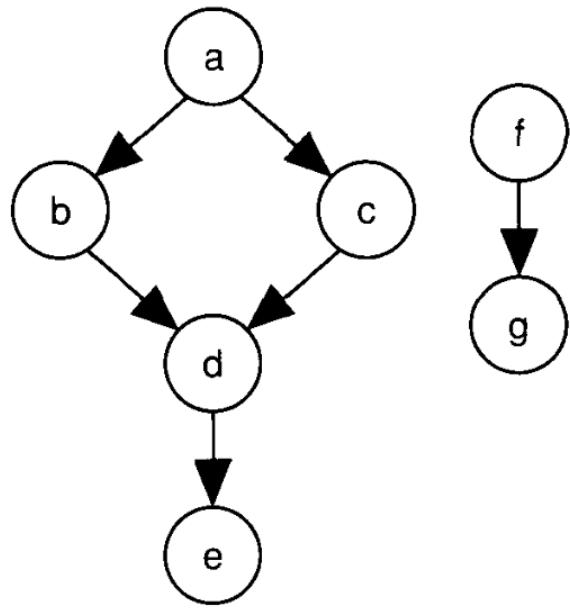
`connessi(a,d)`

`connessi(a,d) :- arco(a,Z1), connessi(Z1,d).`  
{ X<sub>1</sub>=a , Y<sub>1</sub>=d }

Figure 2.4 A simple graph



## Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`  
non unifica

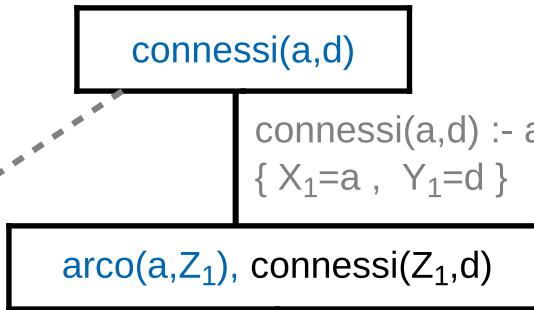
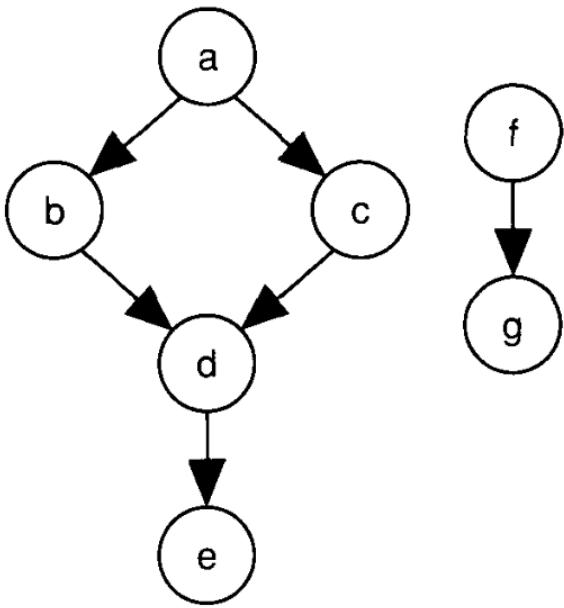


Figure 2.4 A simple graph



## Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`  
non unifica

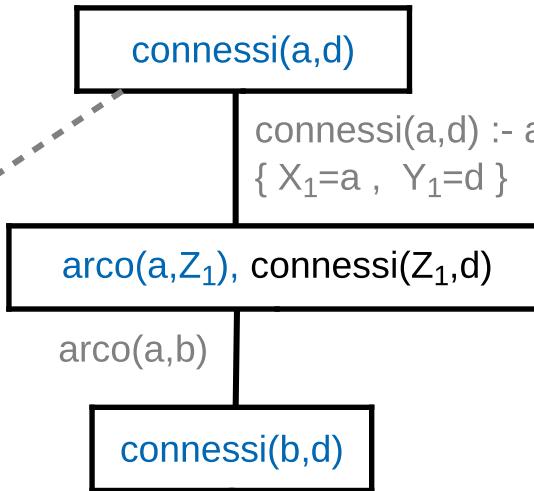
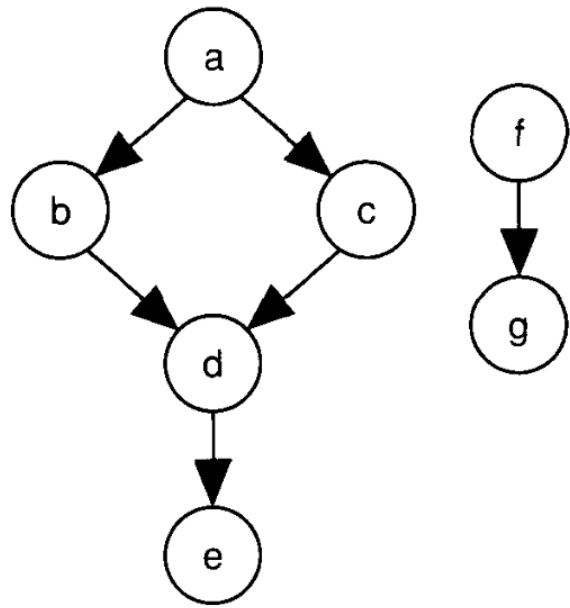


Figure 2.4 A simple graph



## Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`  
non unifica

`connessi(X,X)`  
non unifica

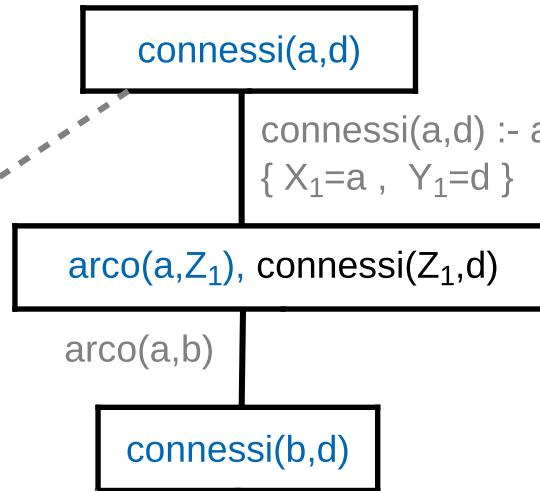
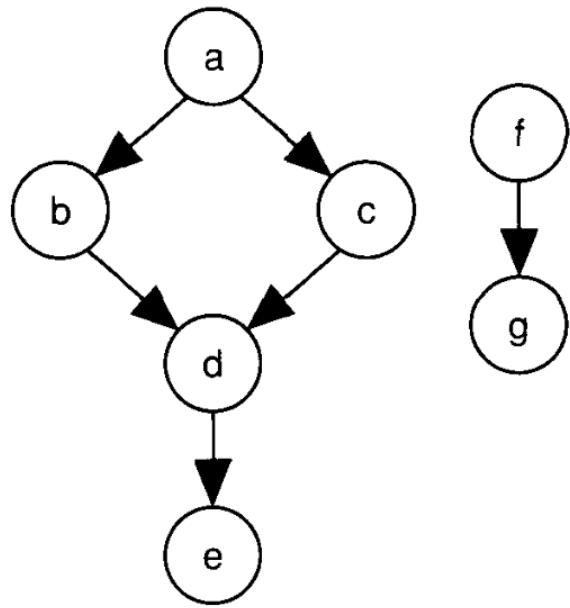


Figure 2.4 A simple graph



## Search tree parziale per la query `connessi(a,d)`



*connessi(X,X)  
non unifica*

*connessi(X,X)  
non unifica*

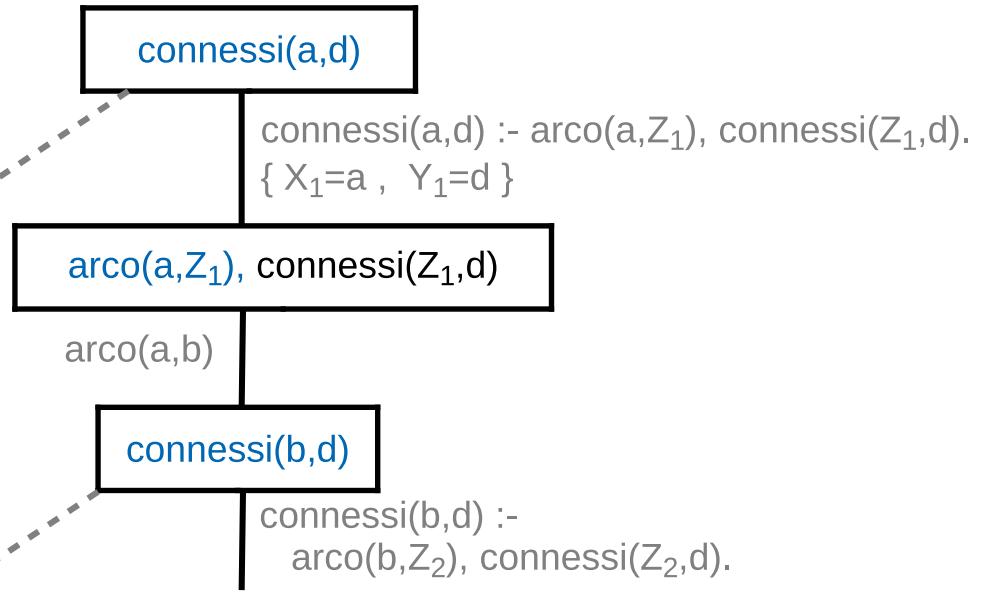


Figure 2.4 A simple graph



## Search tree parziale per la query `connessi(a,d)`

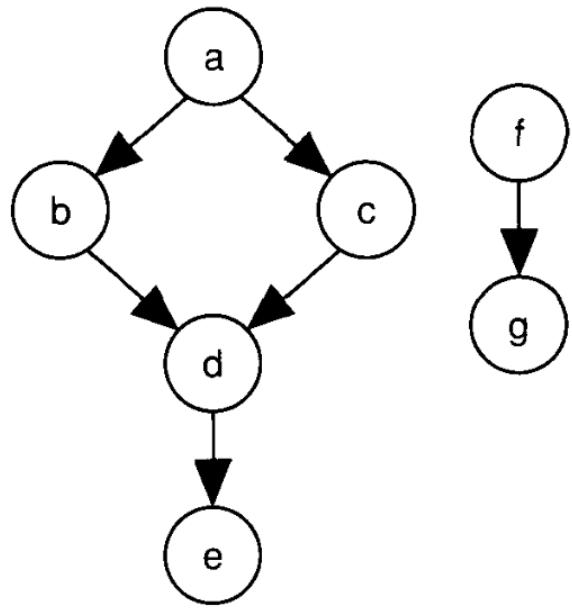
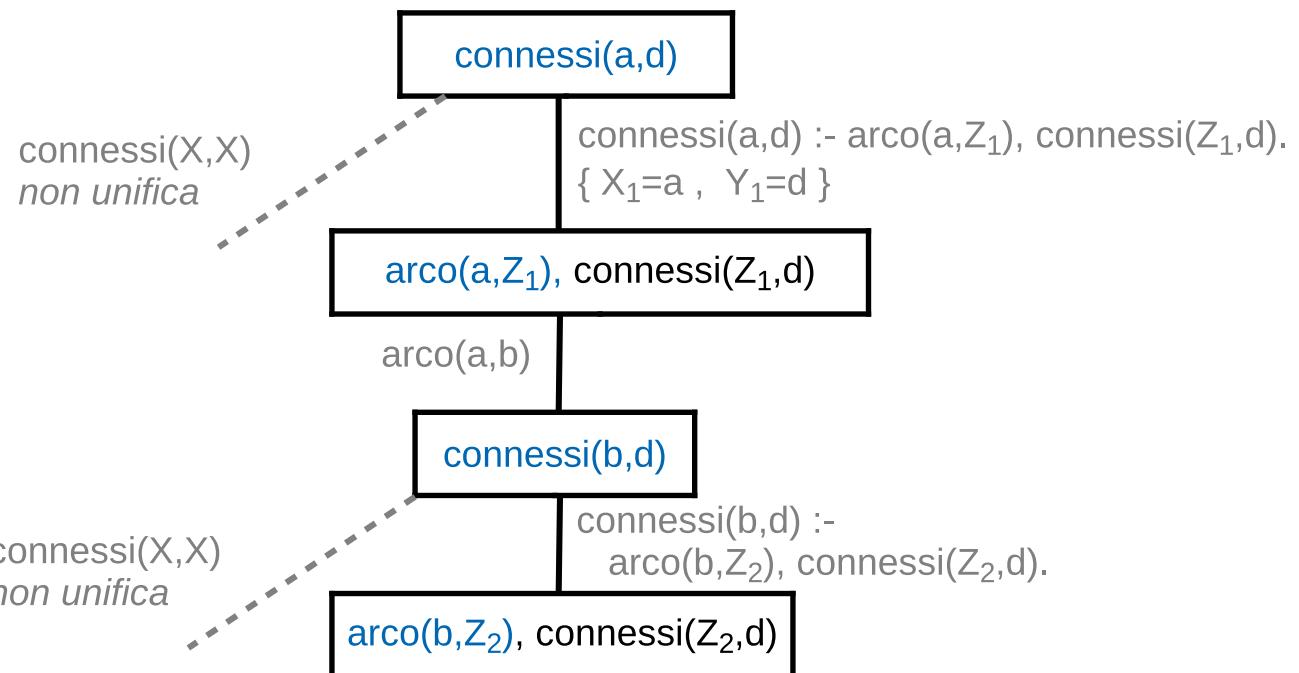


Figure 2.4 A simple graph





## Search tree parziale per la query `connessi(a,d)`

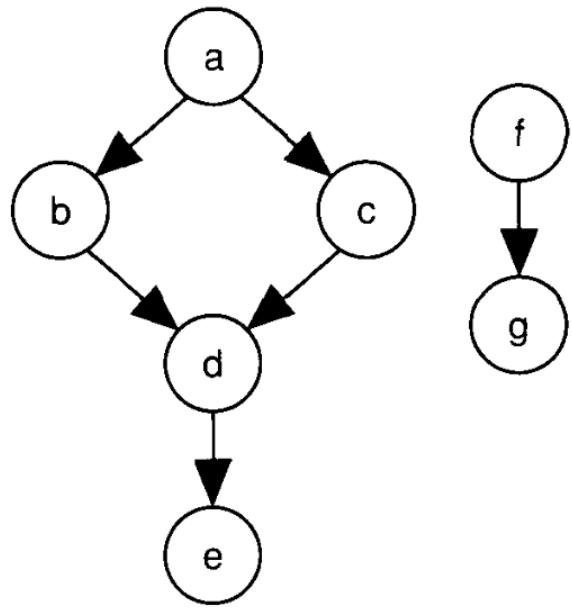
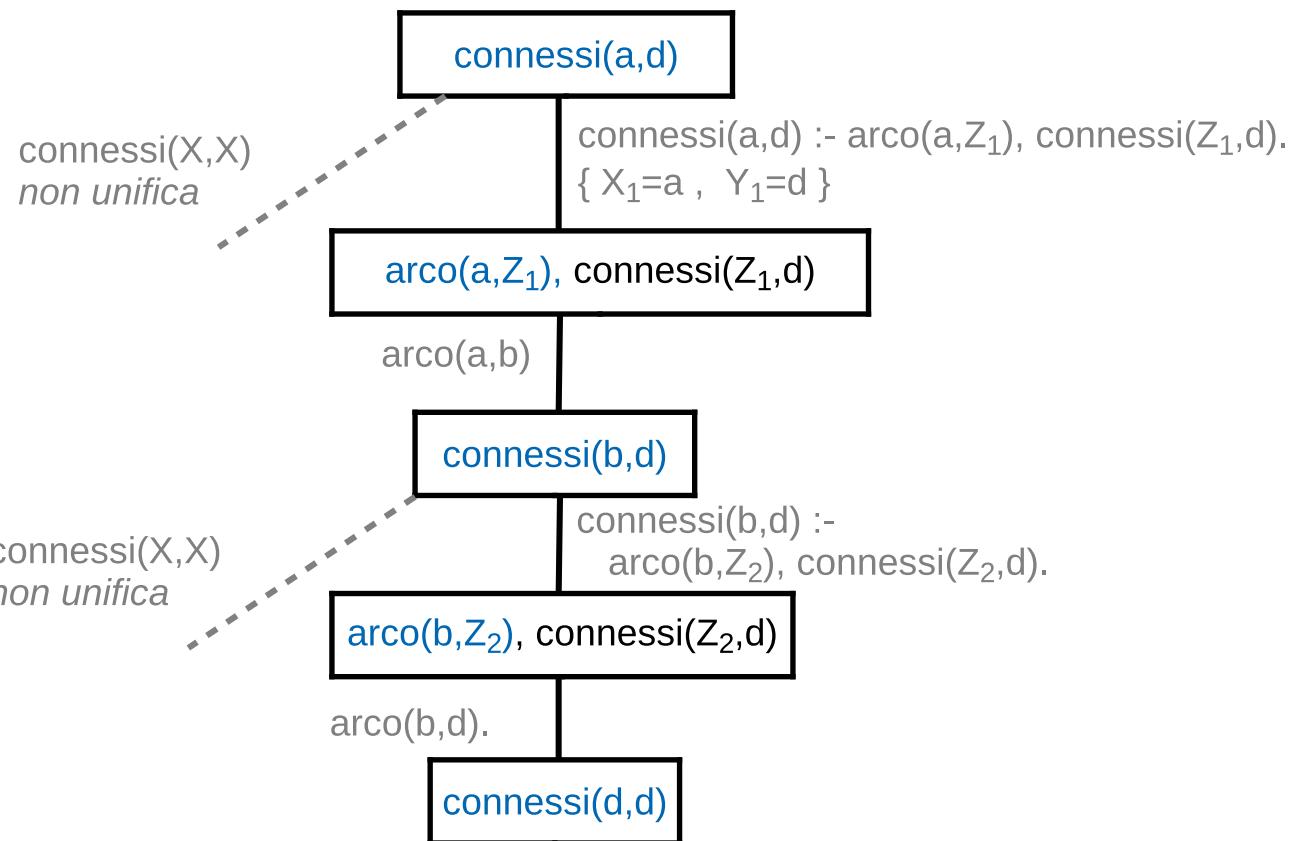


Figure 2.4 A simple graph





## Search tree parziale per la query `connessi(a,d)`

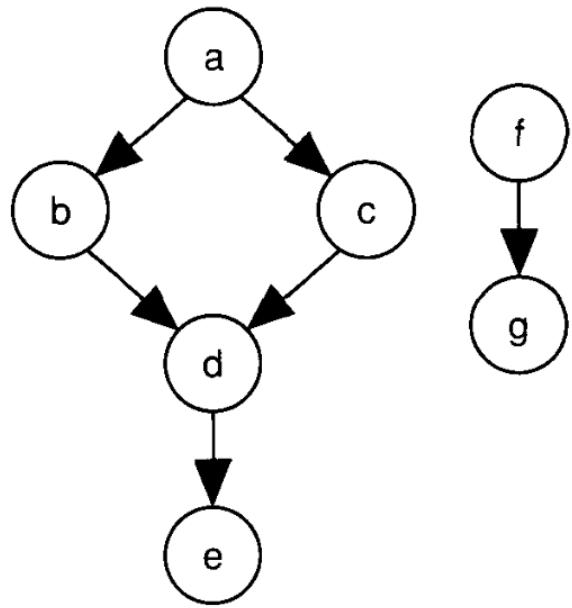
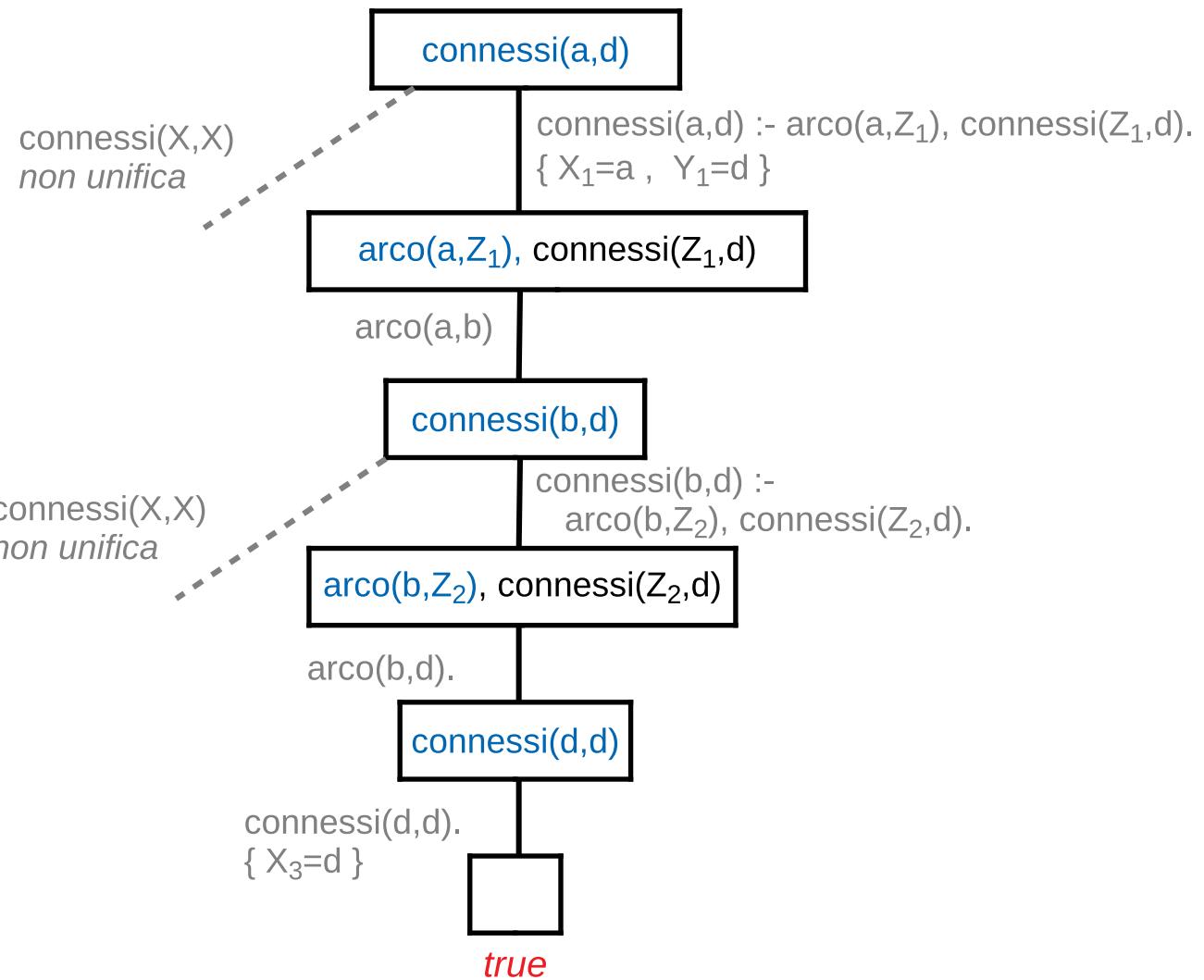


Figure 2.4 A simple graph



Non cerca altre soluzioni perchè la query è ground (inutile restituire altri “true” )



## Esempio di search tree per la query $\text{connessi}(d,f)$

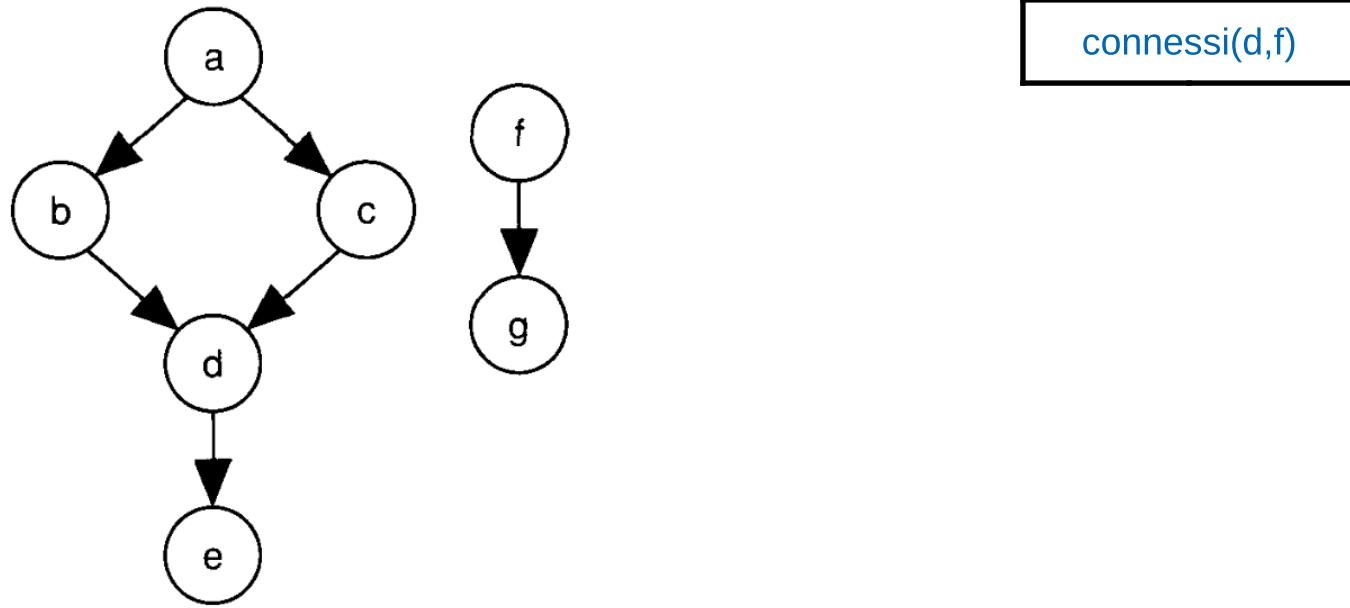


Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$

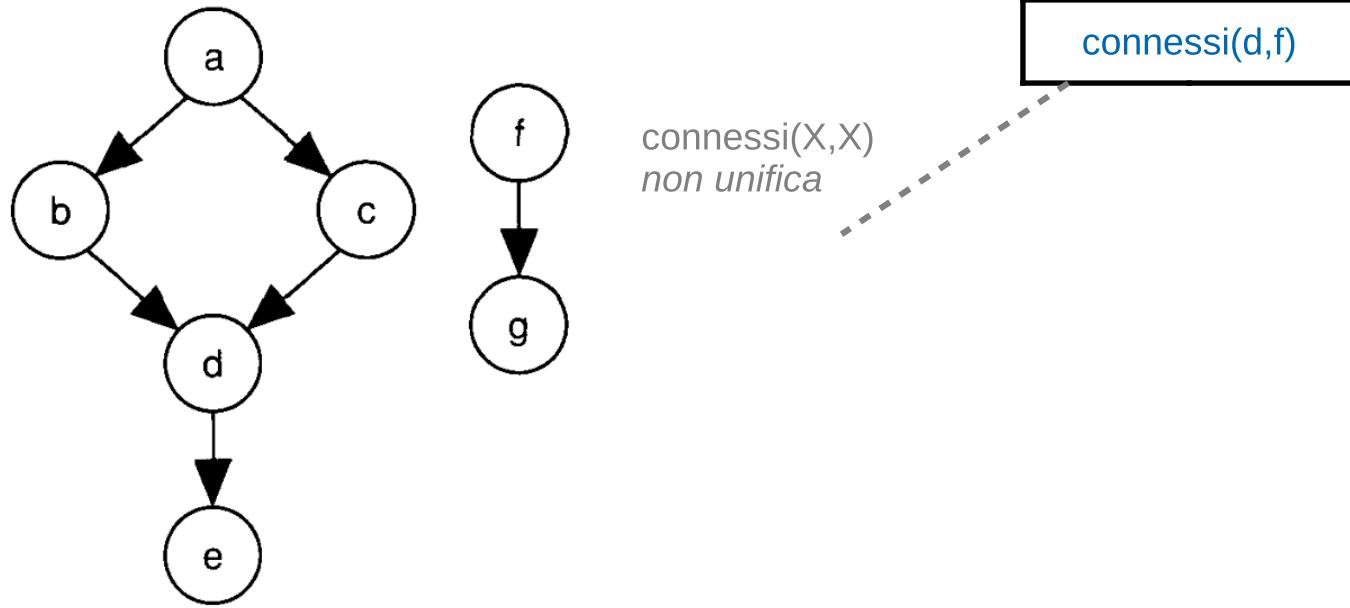
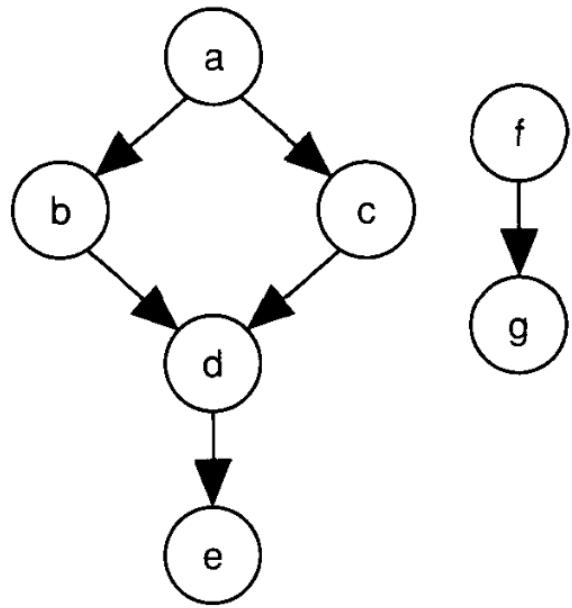


Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$   
non unifica

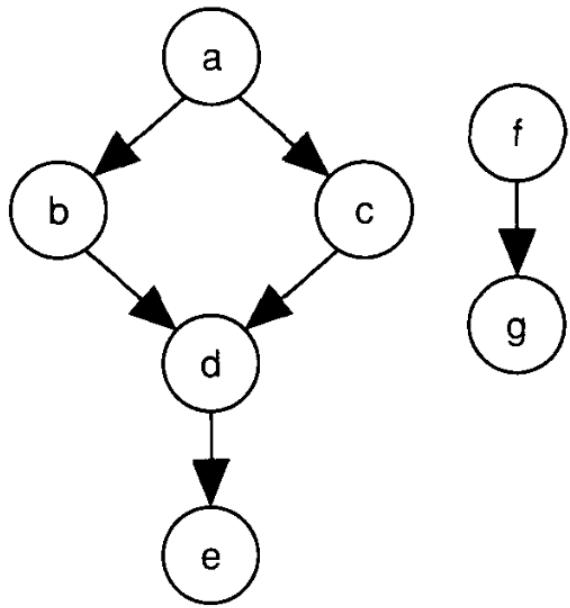
$\text{connessi}(d,f)$

$\text{connessi}(d,f) :- \text{arco}(d,Z_1), \text{connessi}(Z_1,f).$   
 $\{ X_1=d , Y_1=f \}$

Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$   
non unifica

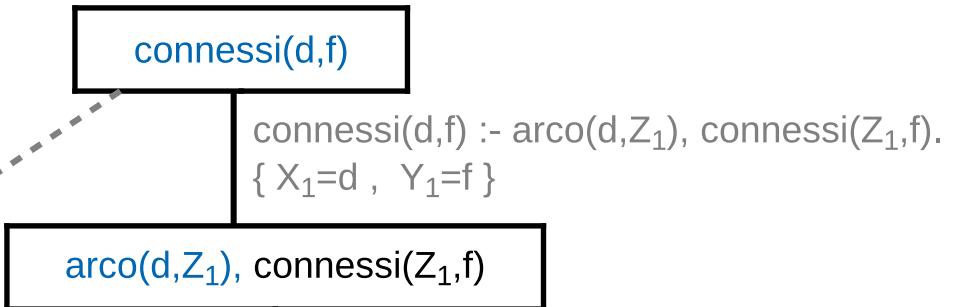
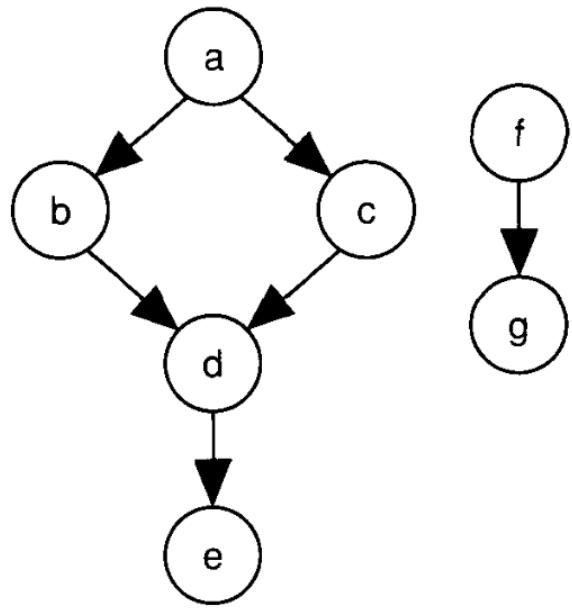


Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$   
non unifica

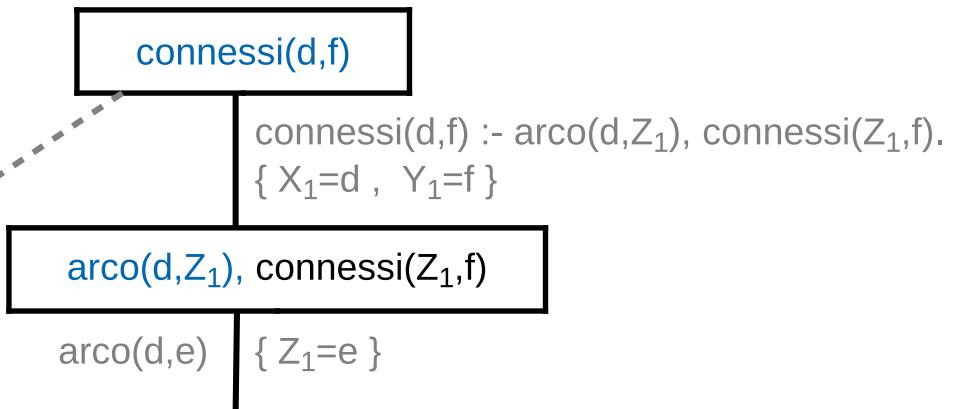
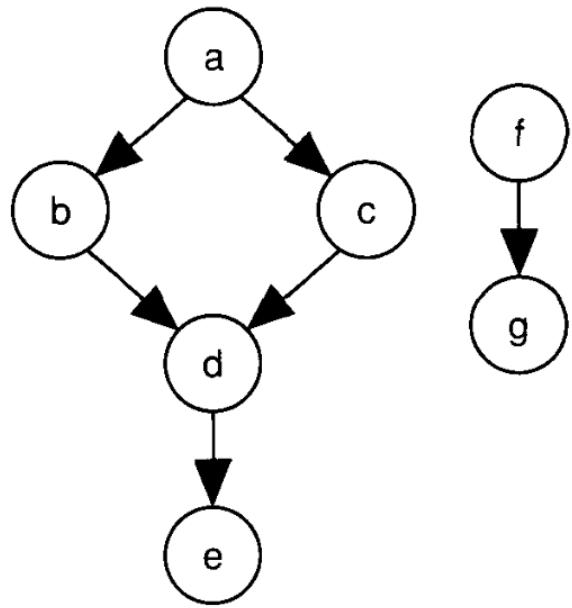


Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$   
non unifica

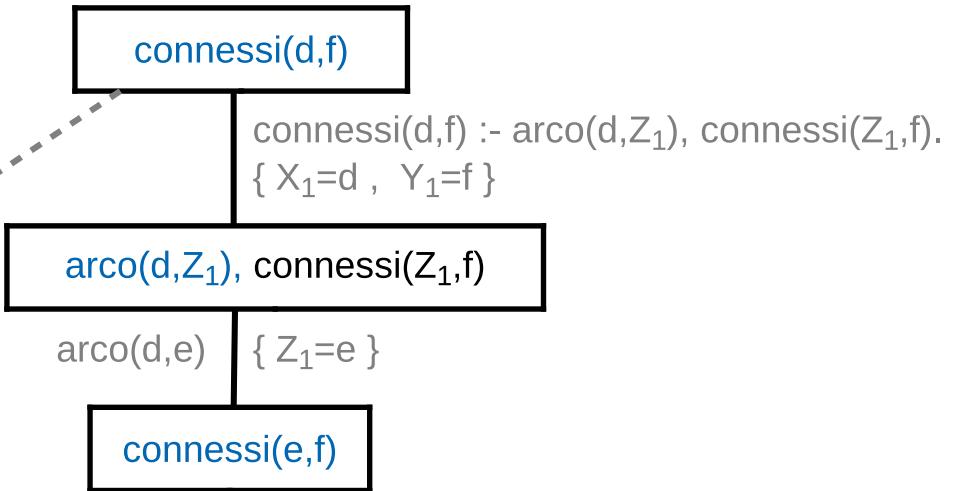
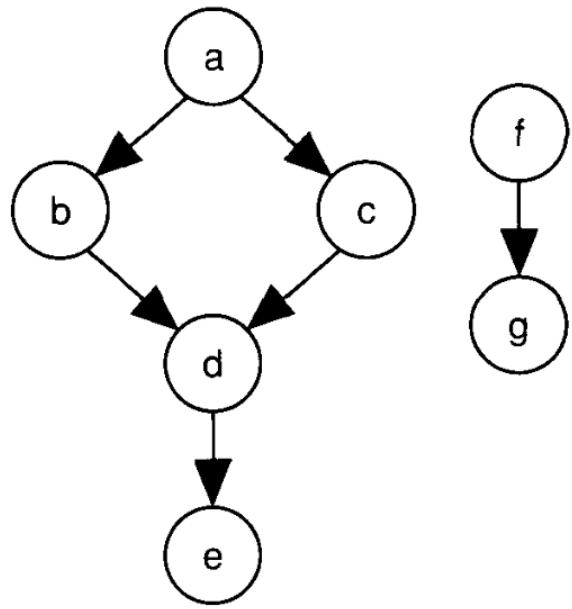


Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$   
non unifica

$\text{connessi}(X,X)$   
non unifica

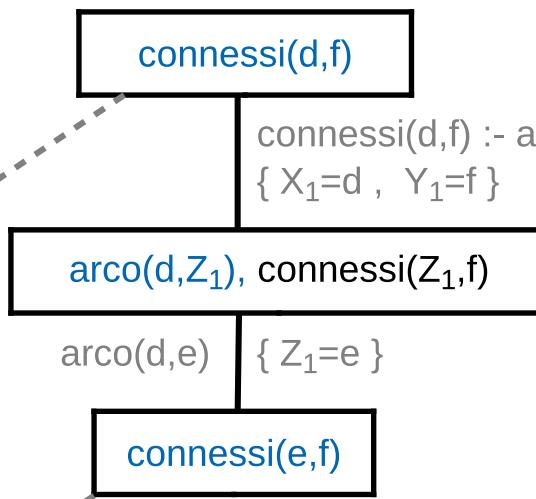
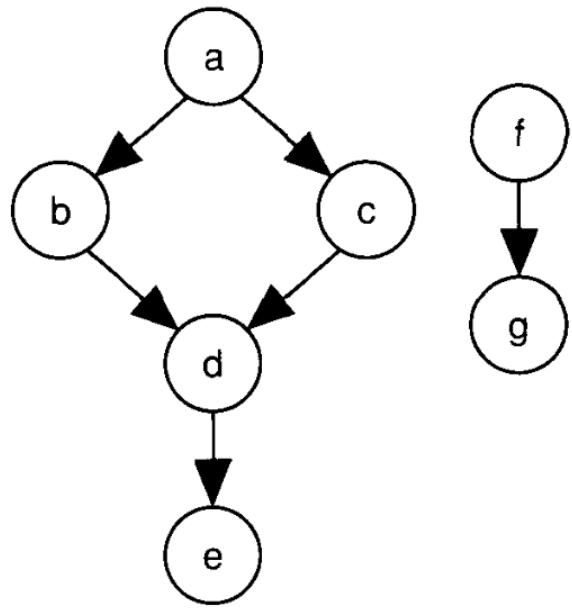


Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$   
non unifica

$\text{connessi}(X,X)$   
non unifica

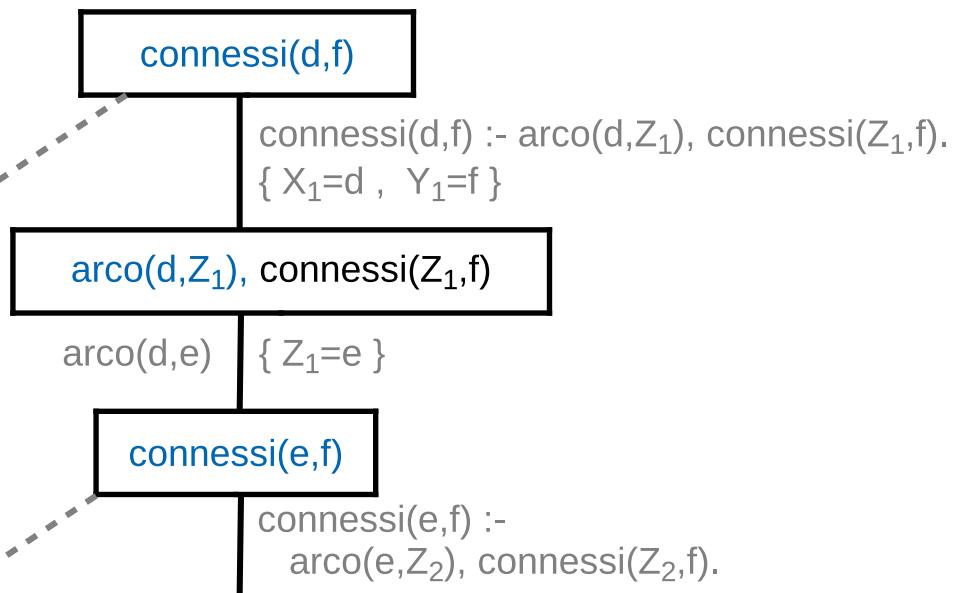


Figure 2.4 A simple graph



## Esempio di search tree per la query $\text{connessi}(d,f)$

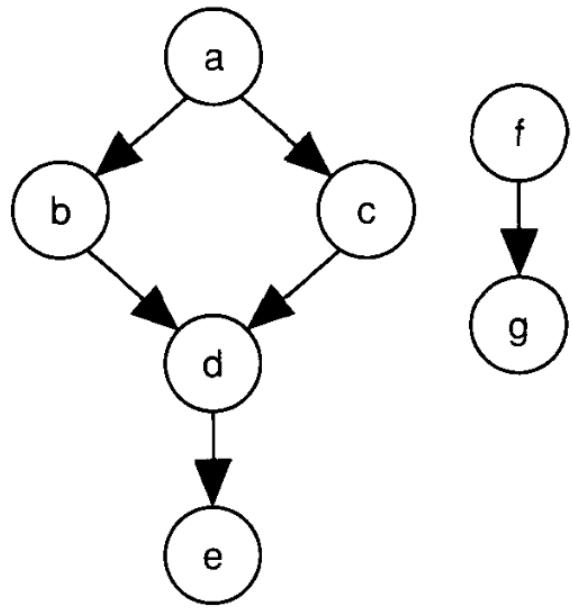
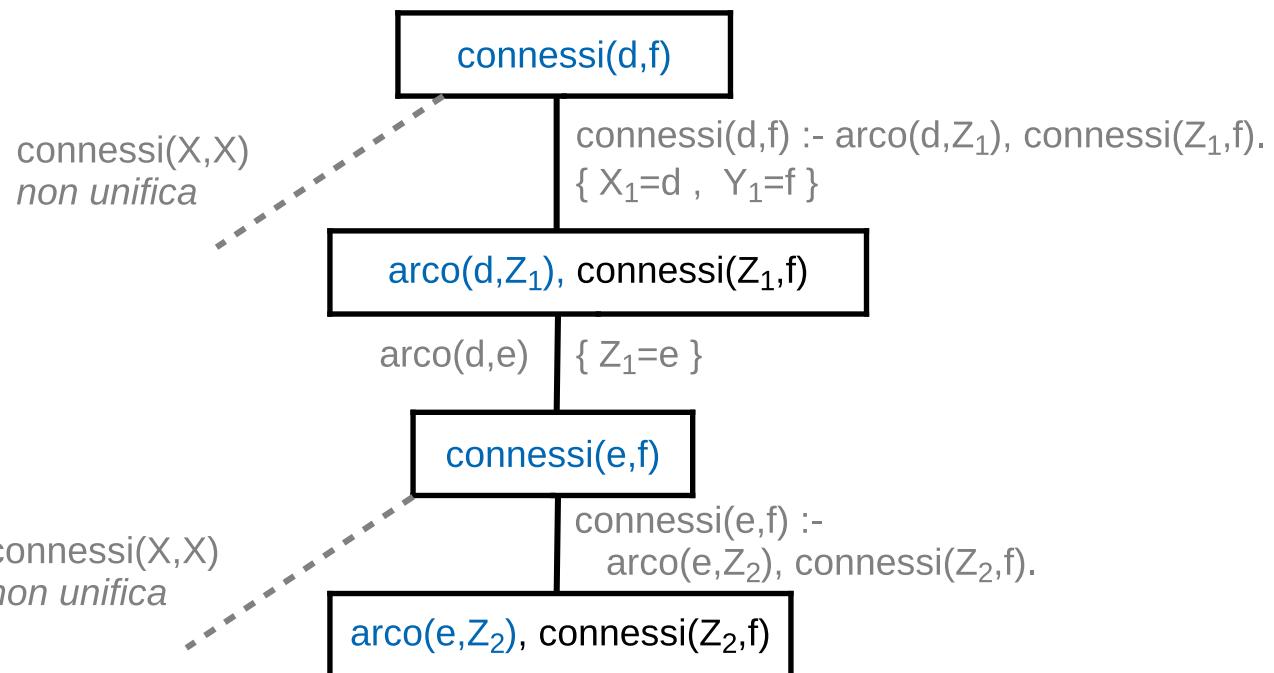


Figure 2.4 A simple graph





## Esempio di search tree per la query $\text{connessi}(d,f)$

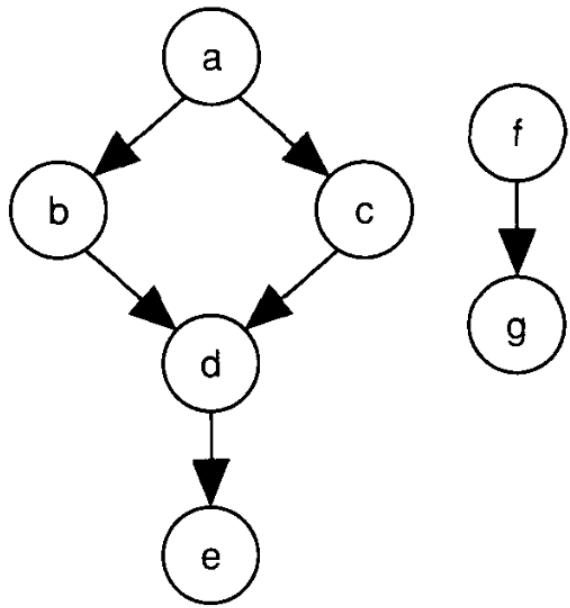
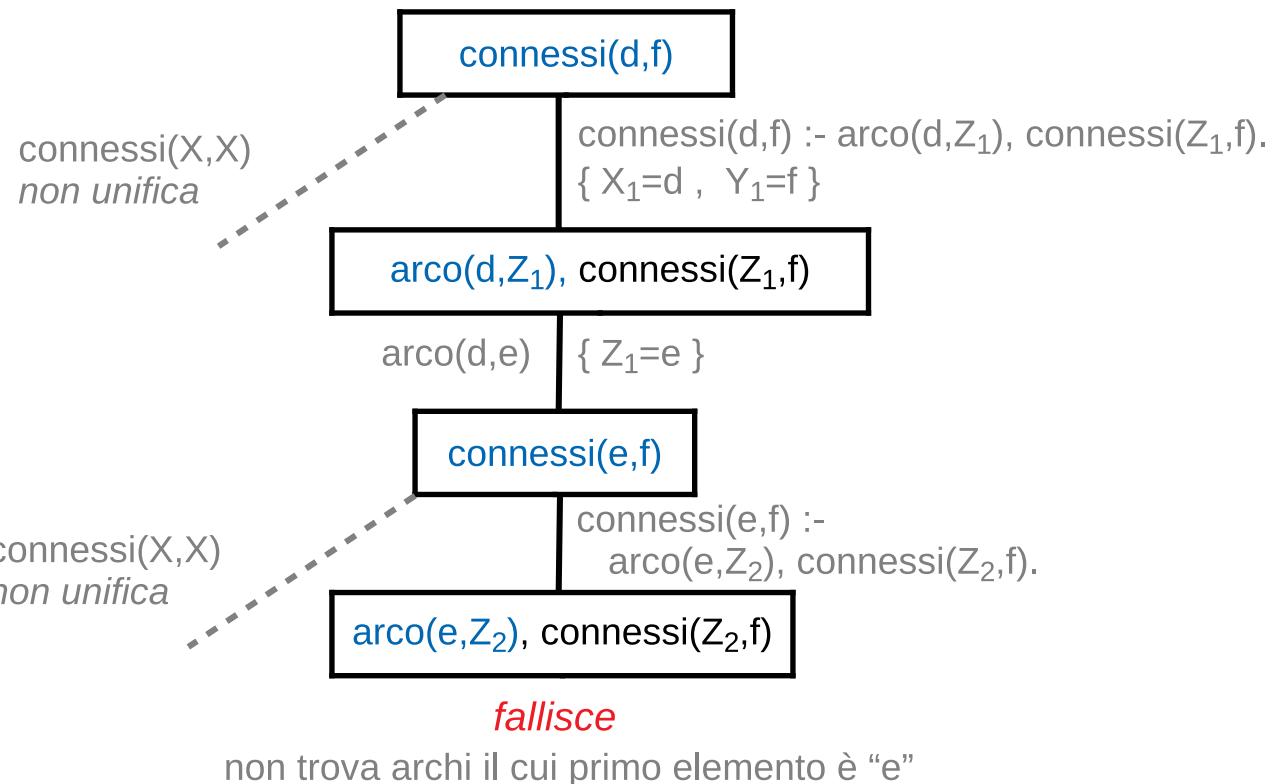


Figure 2.4 A simple graph





## Esempio di search tree per la query $\text{connessi}(d,f)$

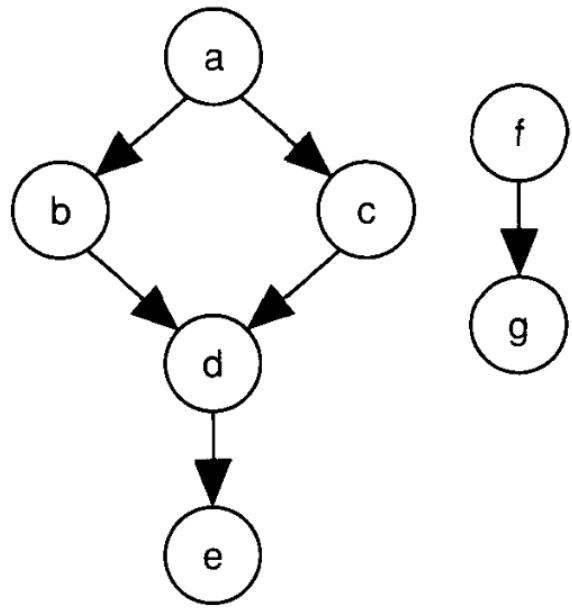
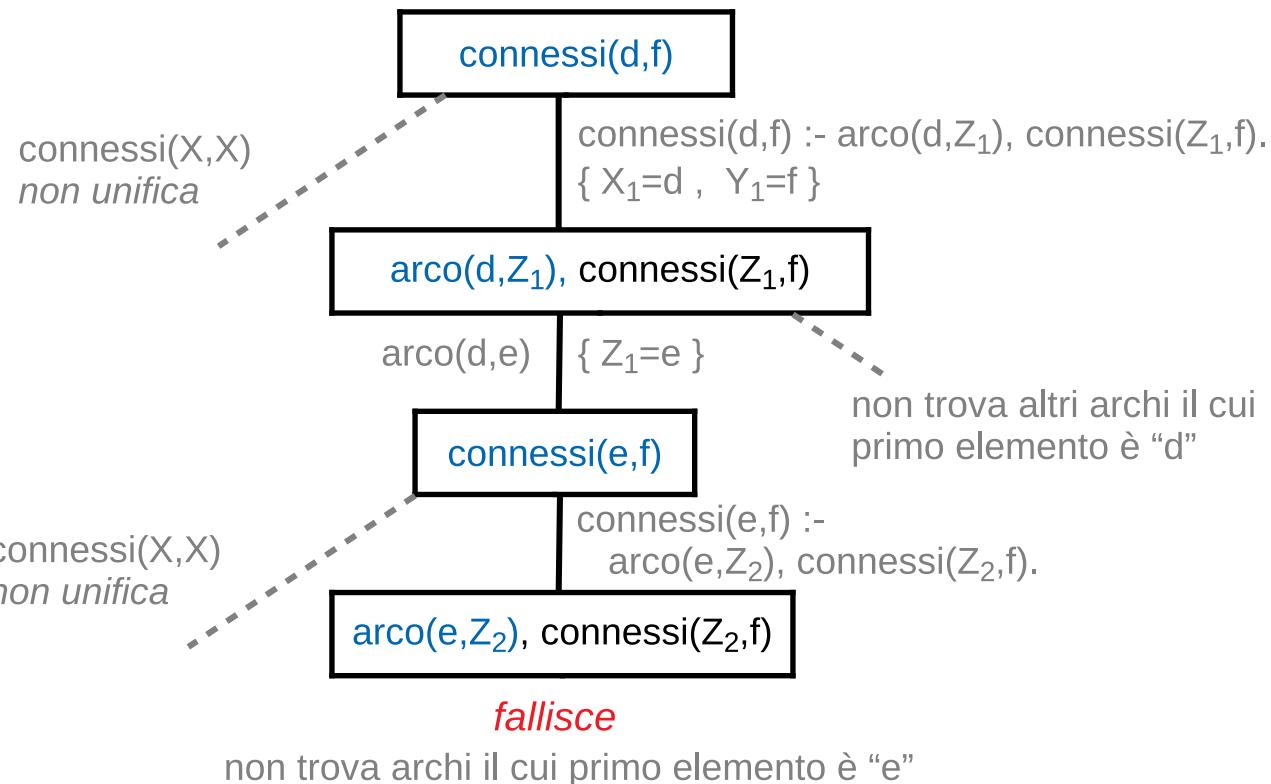


Figure 2.4 A simple graph





## Esempio di search tree per la query $\text{connessi}(d,f)$

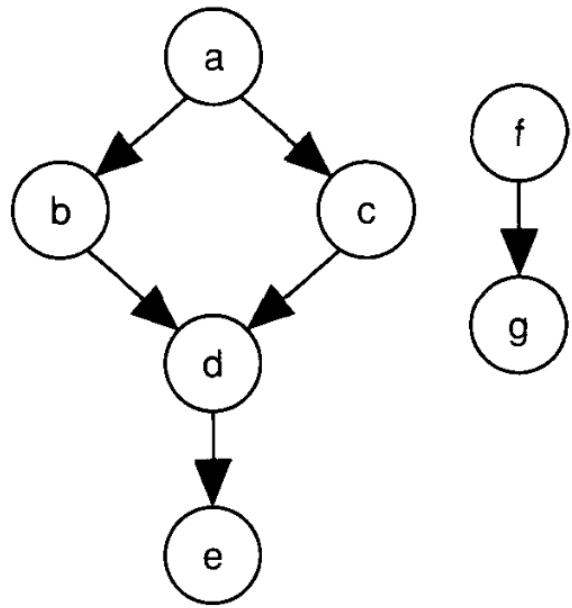
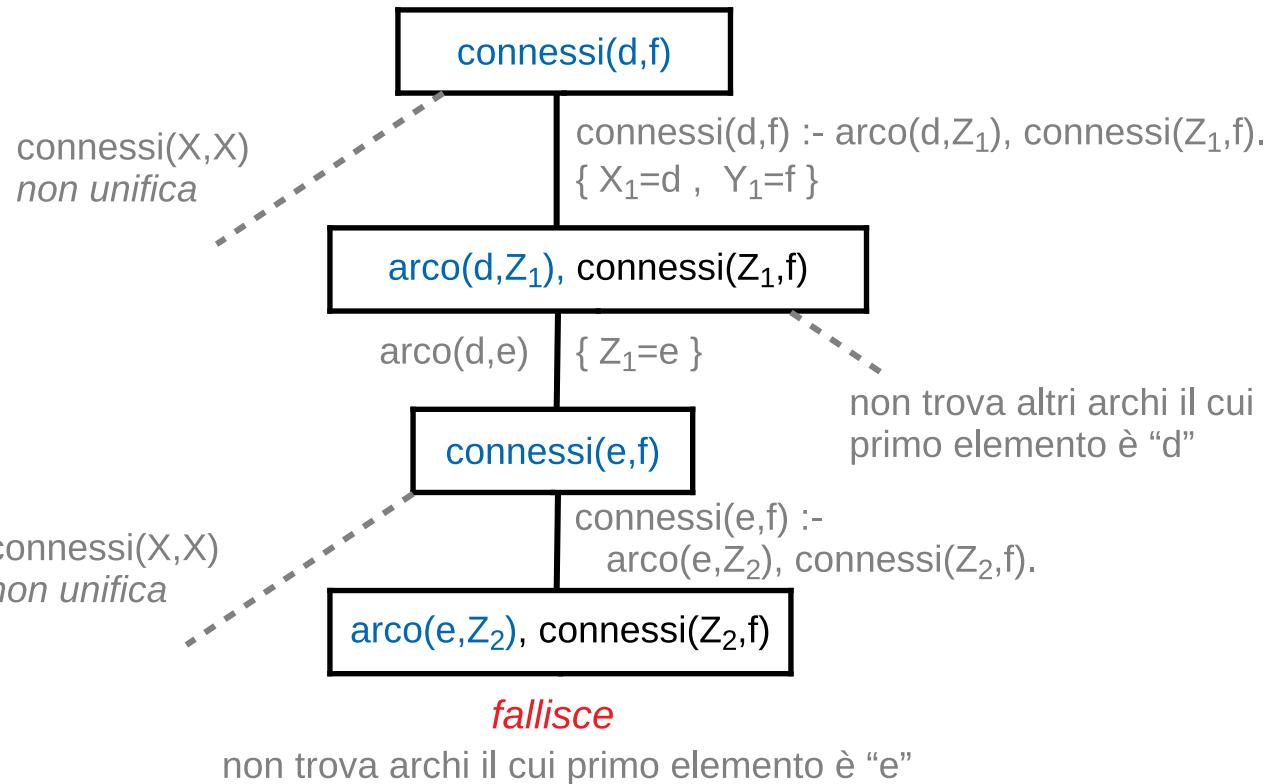


Figure 2.4 A simple graph



*La risposta è false*



## Esempio simile: gli antenati

- Aggiungiamo ora al programma sulla genealogia biblica la nozione di *antenato* (*ancestor*)
- Gli antenati di X sono i suoi genitori, nonni, bisnonni, ...

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

**Cammini su grafi**

Relational algebra

Negazione

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog



## Esempio simile: gli antenati

- Aggiungiamo ora al programma sulla genealogia biblica la nozione di *antenato* (*ancestor*)
- Gli antenati di X sono i suoi genitori, nonni, bisnonni, ...
- Definizione ricorsiva: X è un antenato di Y **se** vale una di queste condizioni:
  1. X è genitore di Y (caso base)
  2. X è genitore di Z e Z è antenato di Y (per qualche Z)

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

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

**Cammini su grafi**

Relational algebra

Negazione

Liste

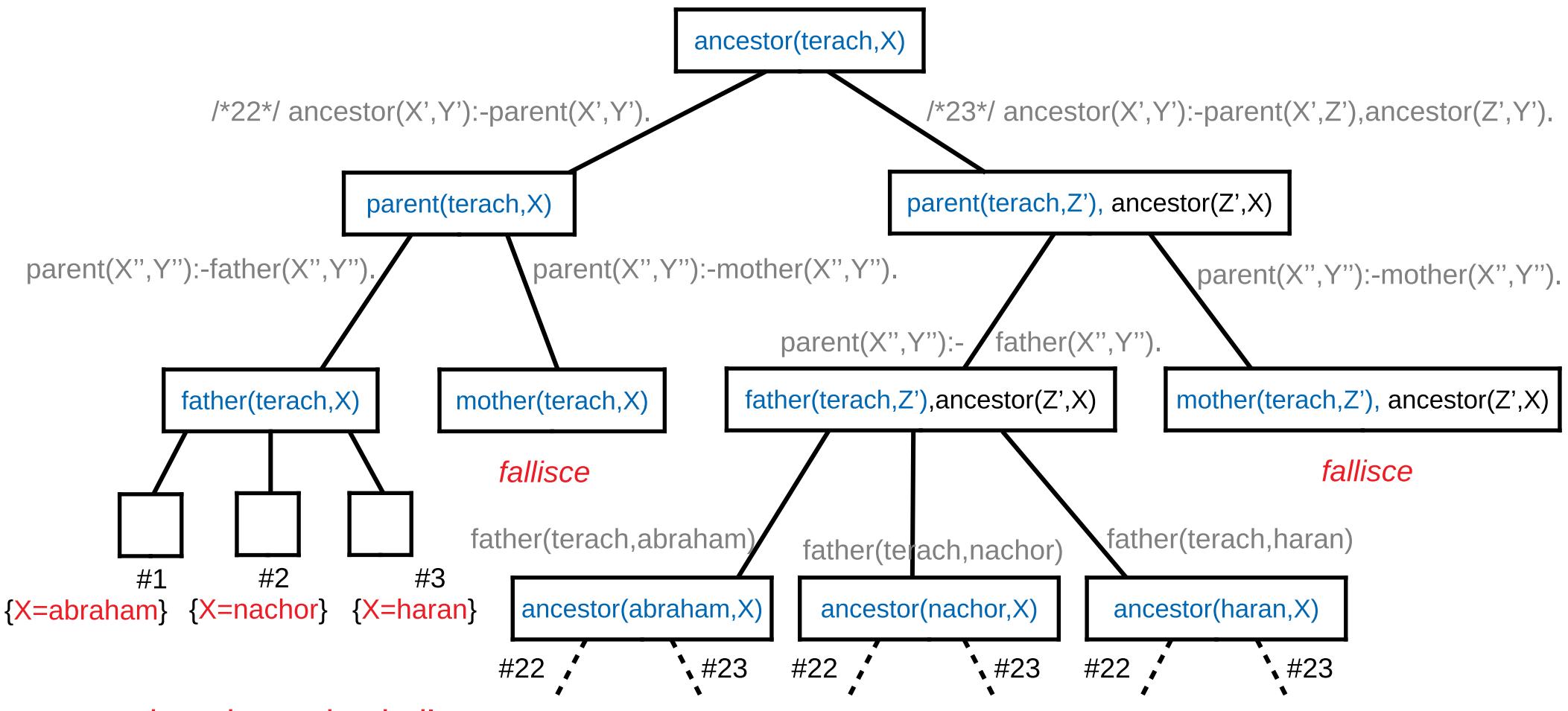
Applicazioni

Programmazione nondeterministica

Unicità di Prolog



## Esempio di search tree per ancestor



continua in modo simile:  
lega X ai discendenti di  
abraham, nachor e haran



# Prolog e l'algebra relazionale

- Ogni tabella relazionale  $r$  si può rappresentare con dei fatti

$r :$

|       |       |       |       |
|-------|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ |
| $a_2$ | $b_2$ | $c_2$ | $d_2$ |
| $a_3$ | $b_3$ | $c_3$ | $d_3$ |
| :     | :     | :     | :     |
| :     | :     | :     | :     |

```
r(a1, b1, c1, d1).
r(a2, b2, c2, d2).
r(a3, b3, c3, d3).
. . .
```

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

**Relational algebra**

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



# Prolog e l'algebra relazionale

- Ogni tabella relazionale  $r$  si può rappresentare con dei fatti

$r :$

|       |       |       |       |
|-------|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ |
| $a_2$ | $b_2$ | $c_2$ | $d_2$ |
| $a_3$ | $b_3$ | $c_3$ | $d_3$ |
| :     | :     | :     | :     |
| :     | :     | :     | :     |

```
r(a1, b1, c1, d1).
r(a2, b2, c2, d2).
r(a3, b3, c3, d3).
.
.
.
```

- Con le regole si può facilmente simulare l'algebra relazionale (ovvero le query SQL)
  - ◆ Prolog genera le n-uple della risposta una per una

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



# Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

**Relational algebra**

Negazione

[Liste](#)

[Applicazioni](#)

Programmazione  
nondeterministica

[Unicità di Prolog](#)

```
/* UNIONE di r e s */
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).
```



# Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

**Relational algebra**

Negazione

[Liste](#)

[Applicazioni](#)

Programmazione  
nondeterministica

[Unicità di 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).
```



# Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di 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).
```



# Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di 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.
```



# Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di 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.`

- Scrivere prodotto cartesiano e join su colonne 1 e 2 come esercizio



# Prolog e l'algebra relazionale

- Per la differenza tra relazioni occorre un operatore di **negazione** denotato da `\+1`

```
/* UNIONE di r e s */
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

**Negazione**

[Liste](#)

[Applicazioni](#)

Programmazione  
nondeterministica

[Unicità di Prolog](#)



# Prolog e l'algebra relazionale

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- Per la differenza tra relazioni occorre un operatore di **negazione** denotato da  $\text{\textbackslash}+$ <sup>1</sup>

```
/* UNIONE di r e s */
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).
```

- La negazione trasforma false in true, ovvero fallimenti in successi.
  - ◆  $\text{\textbackslash}+ p(X_1, \dots, X_n)$  è *true* se tutti i rami del suo albero di ricerca terminano con un fallimento



# Prolog e l'algebra relazionale

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Relational algebra

**Negazione**

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Per la differenza tra relazioni occorre un operatore di **negazione** denotato da  $\text{\textbackslash+}$ <sup>1</sup>

```
/* UNIONE di r e s */
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).
```

- La negazione trasforma false in true, ovvero fallimenti in successi.
  - ◆  $\text{\textbackslash+ p}(X_1, \dots, X_n)$  è *true* se tutti i rami del suo albero di ricerca terminano con un fallimento
  - ◆ L'albero deve essere finito. Se un programma cade in una ricorsione infinita ovviamente non termina.

---

<sup>1</sup>Nel libro si usa **not**



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

**Liste**

Sintassi

Il predicato member

IN e OUT

Append

[Applicazioni](#)

Programmazione

nondeterministica

[Unicità di Prolog](#)

# Liste



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

**Sintassi**

Il predicato member

IN e OUT

Append

[Applicazioni](#)

Programmazione  
nondeterministica

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

**Sintassi**

Il predicato member

IN e OUT

Append

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata  
[a]

Costruttori esplicativi  
[a | []]



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Sintassi](#)

Il predicato member

IN e OUT

Append

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]

[a , b]

Costruttori esplicativi

[a | []]

[a | [b | []]]



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

**Sintassi**

Il predicato member

IN e OUT

Append

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]

[a, b]

[a, b, c]

Costruttori espliciti

[a | []]

[a | [b | []] ]

[a | [b | [c | []] ] ]



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

**Sintassi**

Il predicato member

IN e OUT

Append

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]  
[a, b]  
[a, b, c]  
[a | X]  
[a, b | X]

Costruttori espliciti

[a | []]  
[a | [b | []] ]  
[a | [b | [c | []] ] ]  
[a | X]  
[a | [b | X] ]



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

**Sintassi**

Il predicato member

IN e OUT

Append

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

## Abbreviata

[a]  
[a, b]  
[a, b, c]  
[a | X]  
[a, b | X]  
[a, b, c | X]

## Costruttori esplicativi

[a | []]  
[a | [b | []] ]  
[a | [b | [c | []] ] ]  
[a | X]  
[a | [b | X] ]  
[a | [b | [c | X] ] ]



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

**Sintassi**

Il predicato member

IN e OUT

Append

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

## Abbreviata

[a]  
[a, b]  
[a, b, c]  
[a | X]  
[a, b | X]  
[a, b, c | X]

## Costruttori esplicativi

[a | []]  
[a | [b | []] ]  
[a | [b | [c | []] ] ]  
[a | X]  
[a | [b | X] ]  
[a | [b | [c | X] ] ]



# Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

**Sintassi**

Il predicato member

IN e OUT

Append

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]  
[a, b]  
[a, b, c]  
[a | X]  
[a, b | X]  
[a, b, c | X]

Costruttori espliciti

[a | []]  
[a | [b | []] ]  
[a | [b | [c | []] ] ]  
[a | X]  
[a | [b | X] ]  
[a | [b | [c | X] ] ]

■ Notare la possibilità di esprimere liste *parzialmente specificate*, dove alcuni elementi ed eventualmente la coda sono variabili.

[a , **X** , b | **Y**]



## Il predicato member

■ Il predicato member cerca un elemento X in una lista L

- ◆ è ovviamente *ricorsivo*
- ◆ usiamo la wildcard ‘\_’

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Sintassi](#)

[Il predicato member](#)

IN e OUT

Append

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



## Il predicato member

- Il predicato member cerca un elemento X in una lista L

- ◆ è ovviamente *ricorsivo*
  - ◆ usiamo la 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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

**Il predicato member**

IN e OUT

Append

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Esempi di derivazione per member

member(1,[1,2])

```
member(X, [X | _]).
member(X, [_ | L]) :- member(X, L).
```



## Esempi di derivazione per member

member( $X_1, [X_1 | \underline{\quad}]$ ) unifica  
 $\{X_1 = 1\}$

member(1,[1,2])

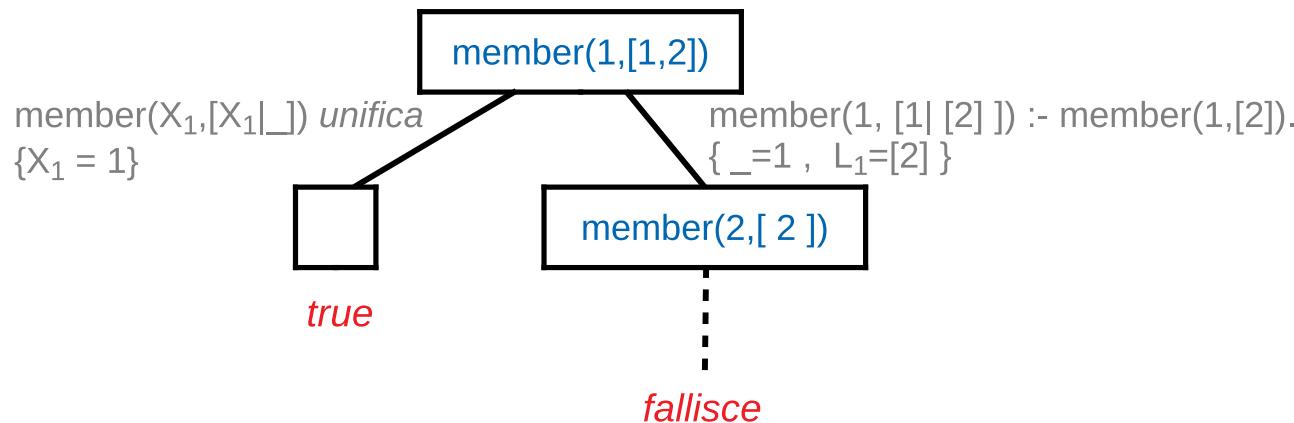


true

```
member(X, [X | _]).
member(X, [_ | L]) :- member(X, L).
```



## Esempi di derivazione per member



*Questo ramo non viene esplorato perché la query è ground*

```
member(X, [X|_]).
member(X, [_|L]) :- member(X,L).
```



## Esempi di derivazione per member

```
member(2,[1,2])
```

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```



## Esempi di derivazione per member

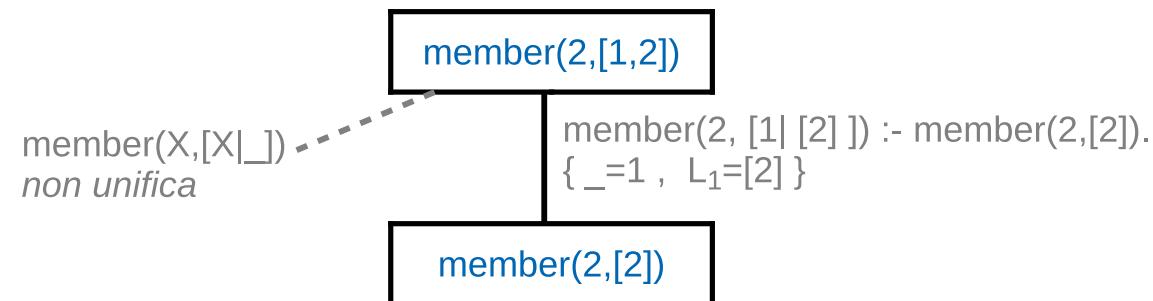
member(2,[1,2])

member(X,[X|\_])  
*non unifica*

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```



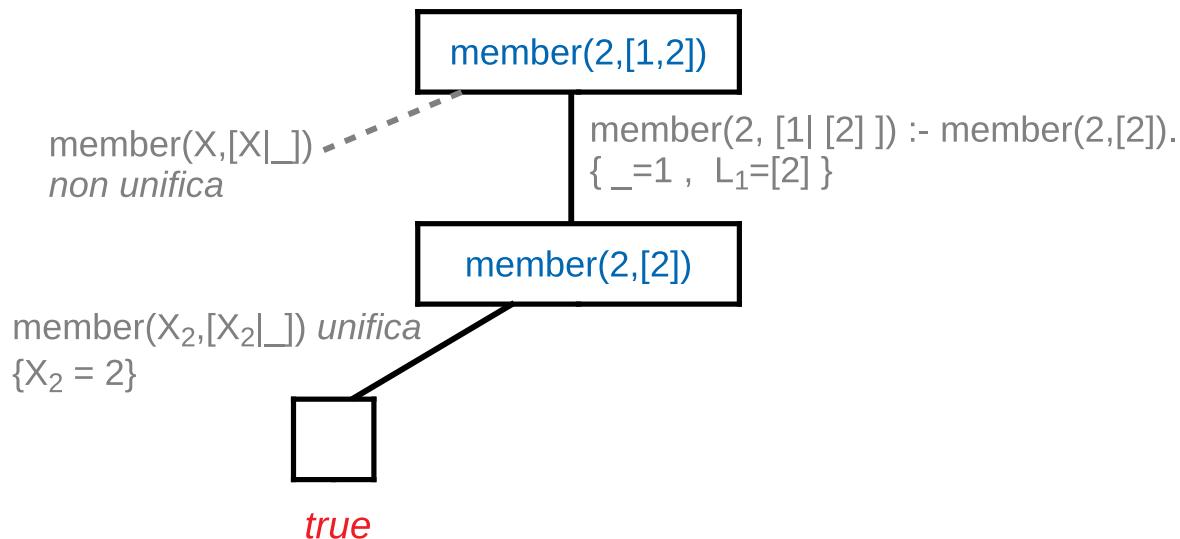
## Esempi di derivazione per member



```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```



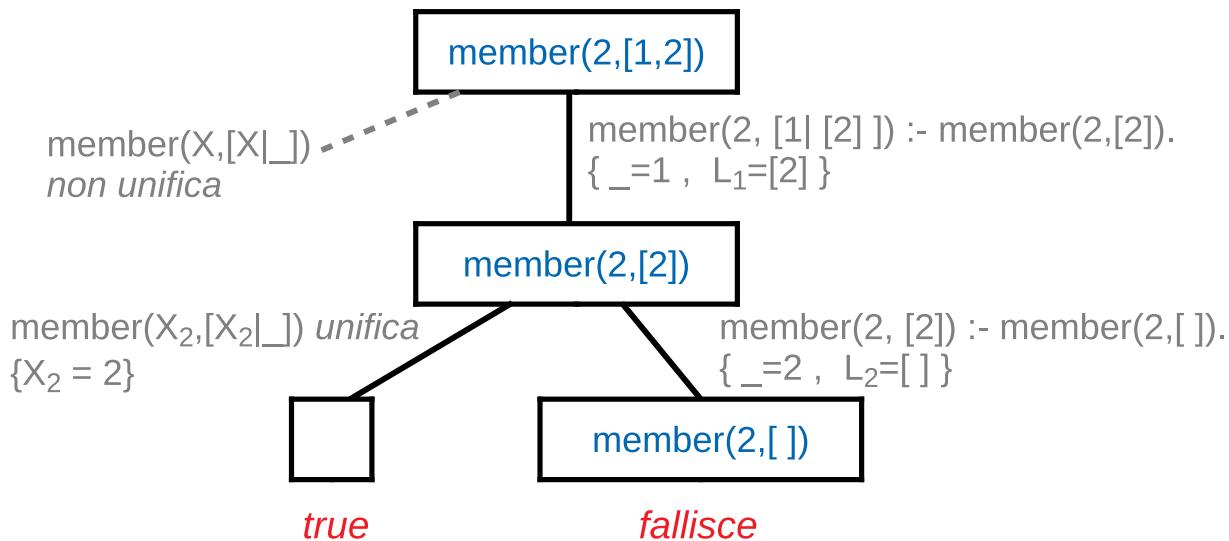
## Esempi di derivazione per member



```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```



## Esempi di derivazione per member

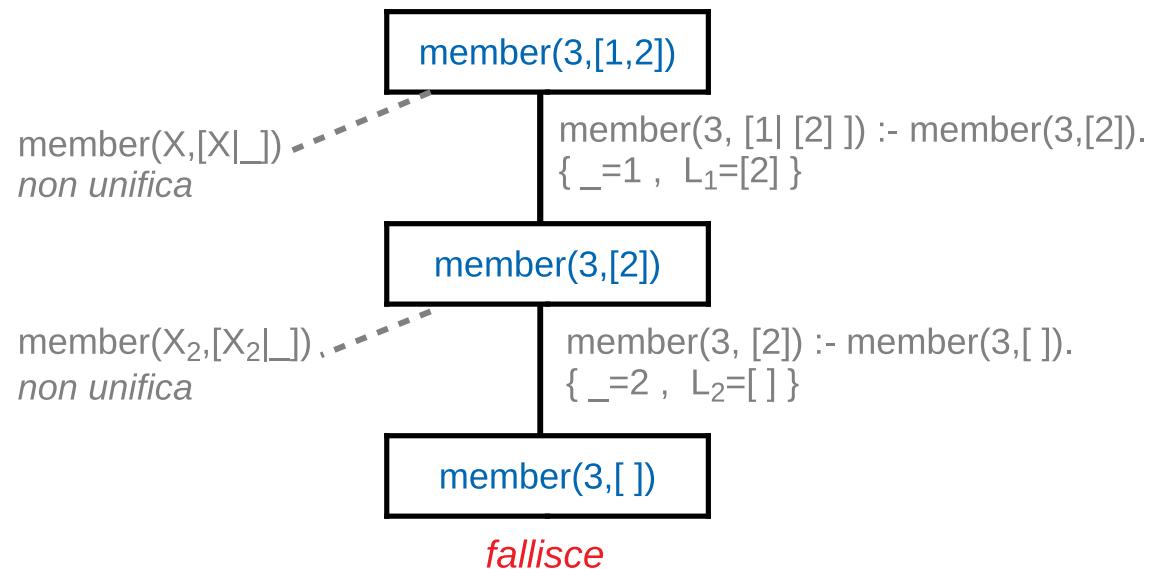


*Questo ramo non viene esplorato perché la query è ground*

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```



# Esempi di derivazione per member

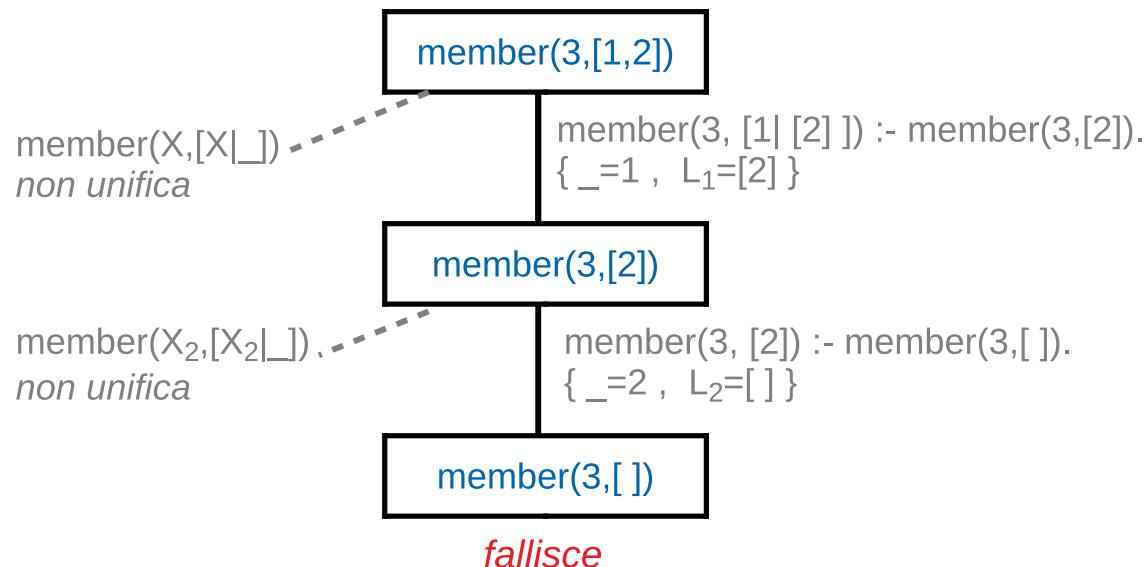


Non ci sono altri rami da provare  
→ Risponde *false*

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```



## Esempi di derivazione per member



Non ci sono altri rami da provare  
→ Risponde *false*

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

- Notare che in ML avrebbe sollevato una eccezione. Qui restituisce *false*



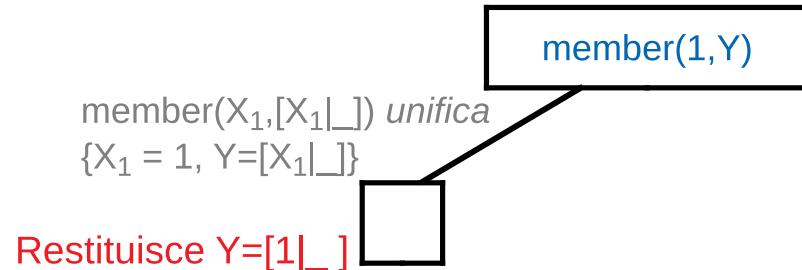
## Esempi di derivazione per member

member(1,Y)

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```



## Esempi di derivazione per member



```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

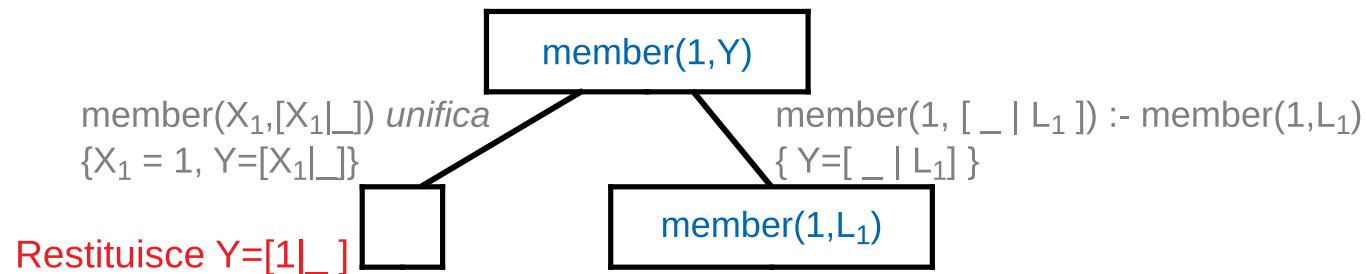
Il valore di Y si ottiene così:

X<sub>1</sub>=1,

Y=[X<sub>1</sub>|\_] = [1|\_]



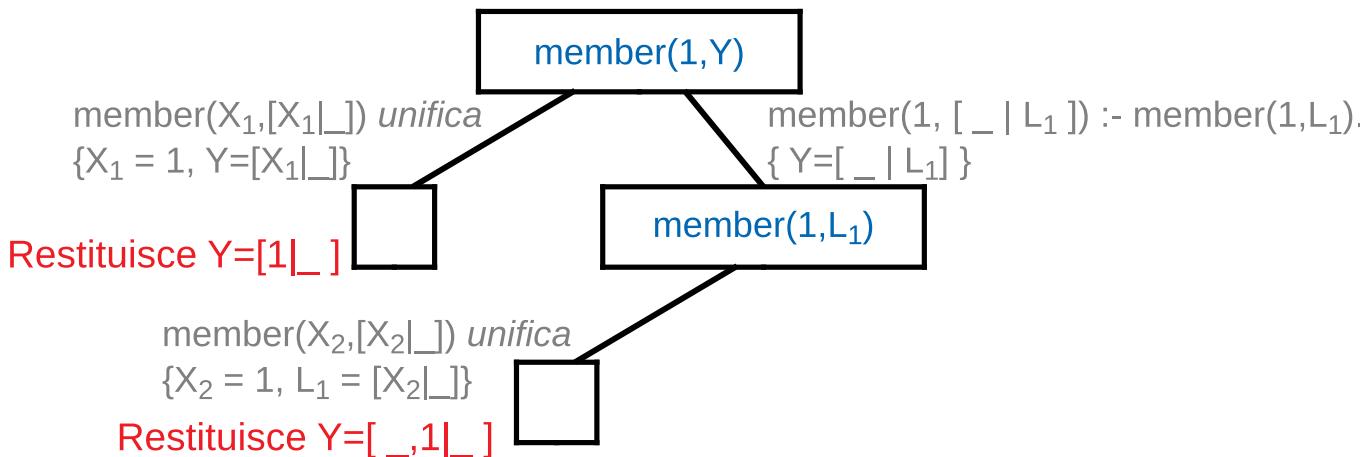
## Esempi di derivazione per member



```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```



## Esempi di derivazione per member



```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

Il valore di Y si ottiene così:

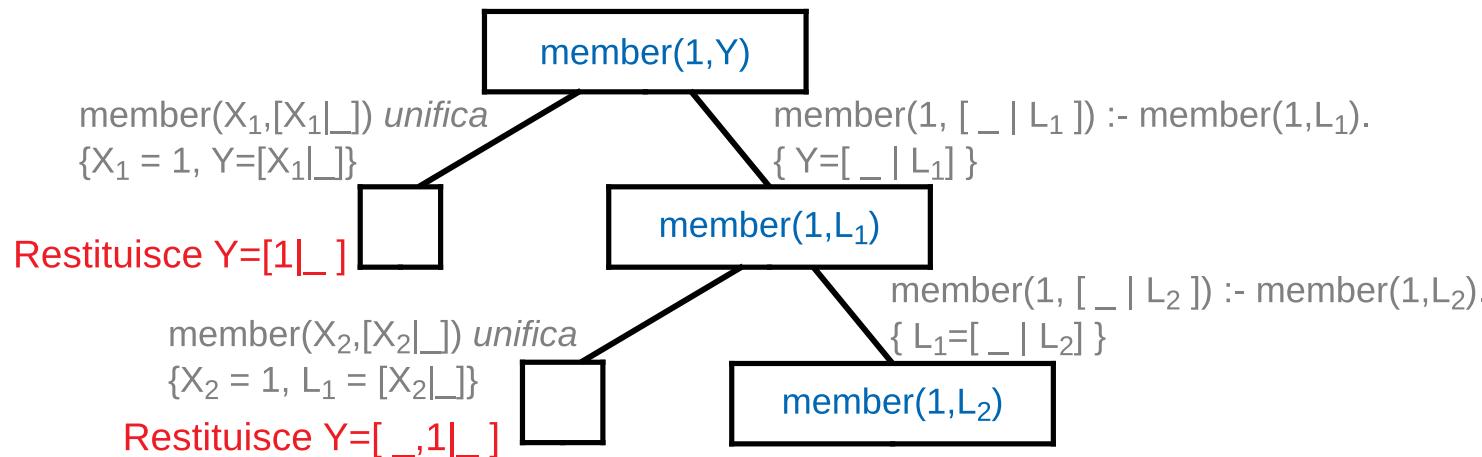
$$X_2 = 1,$$

$$L_1 = [X_2|_] = [1|_],$$

$$Y = [_ | L_1] = [_ | [1|_]] = [_ , 1 | _]$$



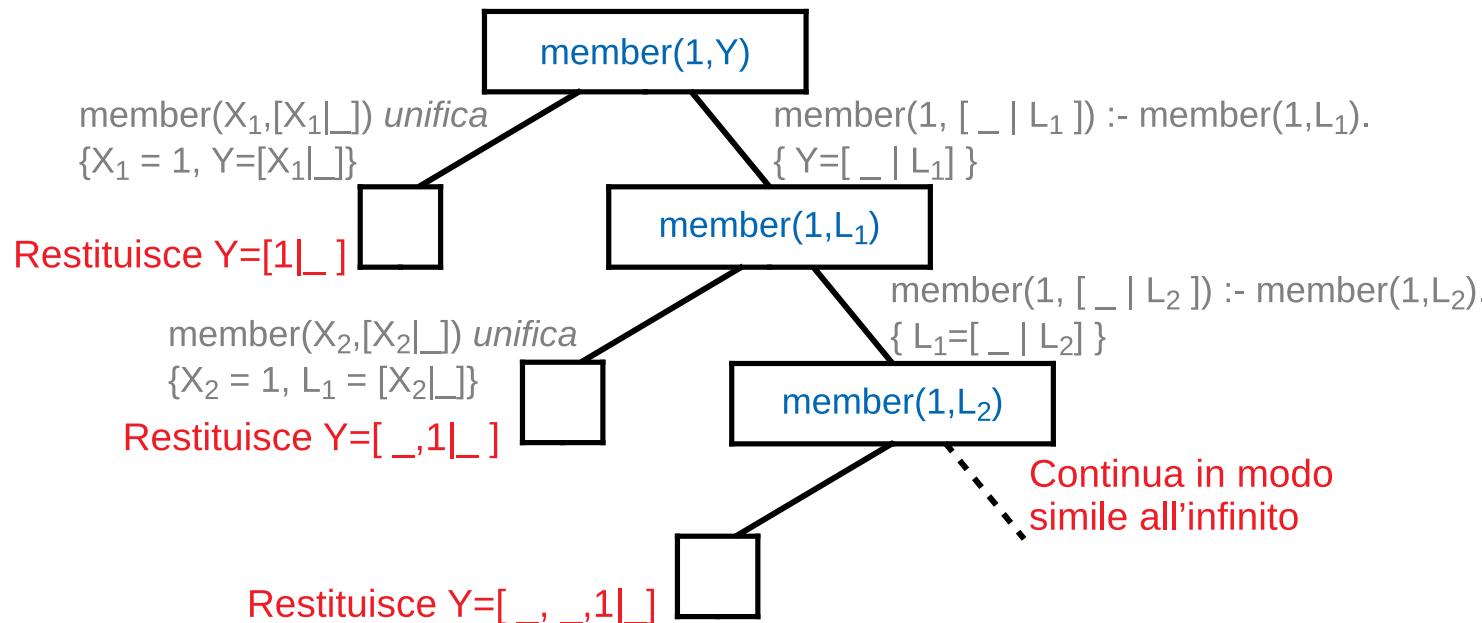
## Esempi di derivazione per member



```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```



# Esempi di derivazione per member



```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

Scrivete voi come si ottiene il valore di Y



# Evanescenza di parametri di Input e Output

- Non c'è una chiara distinzione tra parametri IN e OUT
  - ◆ Ogni parametro può essere legato a un termine con costruttori (input)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

**IN e OUT**

Append

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Evanescenza di parametri di Input e Output

- Non c'è una chiara distinzione tra parametri IN e OUT
  - ◆ Ogni parametro può essere legato a un termine con costruttori (input)
  - ◆ Ogni parametro attuale con una variabile libera produce delle sostituzioni (output)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

**IN e OUT**

Append

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Evanescenza di parametri di Input e Output

- Non c'è una chiara distinzione tra parametri IN e OUT
  - ◆ 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 forniscono sia un input sia un output

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

**IN e OUT**

Append

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Evanescenza di parametri di Input e Output

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

**IN e OUT**

Append

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- Non c'è una chiara distinzione tra parametri IN e OUT
  - ◆ 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 forniscono sia un input sia un output
- Guardiamo gli esempi con member:
  - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)



# Evanescenza di parametri di Input e Output

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

**IN e OUT**

Append

[Applicazioni](#)

Programmazione  
nondeterministica

Unicità di Prolog

- Non c'è una chiara distinzione tra parametri IN e OUT
  - ◆ 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 forniscono sia un input sia un output
- Guardiamo gli esempi con member:
  - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)
  - ◆ `member(<elem. ground>,L)` prende un elemento e restituisce le liste che lo conengono. Modalità: (IN,OUT)



# Evanescenza di parametri di Input e Output

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

**IN e OUT**

Append

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- Non c'è una chiara distinzione tra parametri IN e OUT
  - ◆ 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 forniscono sia un input sia un output
- Guardiamo gli esempi con member:
  - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)
  - ◆ `member(<elem. ground>,L)` prende un elemento e restituisce le liste che lo conengono. Modalità: (IN,OUT)
  - ◆ *Invertibilità dei predicati*: possono rappresentare sia una funzione sia la sua inversa



# Evanescenza di parametri di Input e Output

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

**IN e OUT**

Append

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

- Non c'è una chiara distinzione tra parametri IN e OUT
  - ◆ 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 forniscono sia un input sia un output
- Guardiamo gli esempi con member:
  - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)
  - ◆ `member(<elem. ground>,L)` prende un elemento e restituisce le liste che lo conengono. 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)



## Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

**IN e OUT**

Append

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]
  - (a) scrivere una opportuna query a member che restituisce gli studenti maschi



## Esercizi

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]
  - (a) scrivere una opportuna query a member che restituisce gli studenti maschi
  - (b) disegnare il search tree per la query e la lista data.
  - (c) scrivere le sostituzioni di risposta

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

**IN e OUT**

Append

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



# Esercizi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

**IN e OUT**

Append

Applicazioni

Programmazione  
nondeterministica

Unicità di Prolog

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]
  - (a) scrivere una opportuna query a member che restituisce gli studenti maschi
  - (b) disegnare il search tree per la query e la lista data.
  - (c) scrivere le sostituzioni di risposta
2. Usando member, scrivere la regola per una select (algebra relazionale) sulla relazione r che seleziona i record il cui secondo elemento appartiene a una lista data. Se r ha n colonne, la select ha n+1 argomenti:

```
select_r_2(X1, X2, ..., Xn, Lista)
```



# Esercizi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

**IN e OUT**

Append

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]
  - (a) scrivere una opportuna query a member che restituisce gli studenti maschi
  - (b) disegnare il search tree per la query e la lista data.
  - (c) scrivere le sostituzioni di risposta
2. Usando member, scrivere la regola per una select (algebra relazionale) sulla relazione r che seleziona i record il cui secondo elemento appartiene a una lista data. Se r ha n colonne, la select ha n+1 argomenti:

```
select_r_2(X1, X2, ..., Xn, Lista)
```

3. Scrivere un predicato reverse che inverte una lista. Prendere esempio dalla soluzione per ML che fa uso di un accumulatore per costruire progressivamente la lista invertita:

```
/* significato dei parametri del predicato */
reverse(Lista, ListaInvertita, Accumulatore)
```

Poi usare l'overloading per avere una reverse a 2 soli argomenti.



## Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */
append(Lista1 , Lista2 , Risultato)
```

Ispirarsi alla soluzione ricorsiva per ML

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

**Append**

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */
append(Lista1 , Lista2 , Risultato)
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se  $[a,b,c]$  è un prefisso di una lista data.

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

**Append**

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



## Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */
append(Lista1 , Lista2 , Risultato)
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se [a,b,c] è un prefisso di una lista data.
  - verificare disegnando il search tree per la lista [a,b,c,d]

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

**Append**

[Applicazioni](#)

Programmazione  
nondeterministica

Unicità di Prolog



## Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */
append(Lista1 , Lista2 , Risultato)
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se  $[a,b,c]$  è un prefisso di una lista data.
  - verificare disegnando il search tree per la lista  $[a,b,c,d]$
3. Usare l'invertibilità di append per verificare se  $[a,b,c]$  è un suffisso di una lista data.

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

**Append**

[Applicazioni](#)

Programmazione  
nondeterministica

Unicità di Prolog



## Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */
append(Lista1 , Lista2 , Risultato)
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se  $[a,b,c]$  è un prefisso di una lista data.
  - verificare disegnando il search tree per la lista  $[a,b,c,d]$
3. Usare l'invertibilità di append per verificare se  $[a,b,c]$  è un suffisso di una lista data.
  - verificare disegnando il search tree per la lista  $[a,a,b,c]$

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member  
IN e OUT

**Append**

[Applicazioni](#)

Programmazione  
nondeterministica

[Unicità di Prolog](#)



## Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */
append(Lista1 , Lista2 , Risultato)
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se [a,b,c] è un prefisso di una lista data.
  - verificare disegnando il search tree per la lista [a,b,c,d]
3. Usare l'invertibilità di append per verificare se [a,b,c] è un suffisso di una lista data.
  - verificare disegnando il search tree per la lista [a,a,b,c]
4. Usare l'invertibilità di append per definire un predicato last(L,X) che è vero se X è l'ultimo elemento della lista L.

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

**Append**

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Ancora append

## ■ Generazione di tutti i prefissi / suffissi di una lista

```
prefix(Prefix, List) :- append(Prefix, _, List).
```

```
suffix(Suffix, List) :- append(_, Suffix, List).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

**Append**

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Ancora append

## ■ Generazione di tutti i prefissi / suffissi di una lista

```
prefix(Prefix, List) :- append(Prefix, _, List).

suffix(Suffix, List) :- append(_, Suffix, List).
```

## ■ Definizione alternativa di member attraverso append

```
member(X, L) :- append(_, [X|_], L).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

**Append**

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



# Ancora append

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

**Append**

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

## ■ Generazione di tutti i prefissi / suffissi di una lista

```
prefix(Prefix, List) :- append(Prefix, _, List).

suffix(Suffix, List) :- append(_, Suffix, List).
```

## ■ Definizione alternativa di member attraverso append

```
member(X, L) :- append(_, [X|_], L).
```

## ■ Calcolo delle sottoliste di L. Notare che S è una sottolista di L sse valgono queste due 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).
```



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[\*\*Applicazioni\*\*](#)

Derivate

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## Calcolo simbolico in Prolog



# Calcolo simbolico delle derivate

- In Prolog gli operatori aritmetici sono *costruttori*
  - ◆  $3+5*2$  *non* è uguale a 13!

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

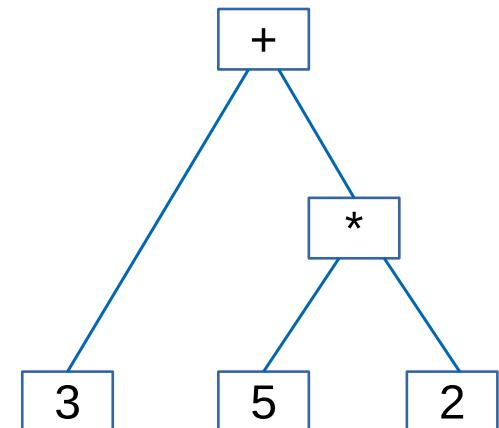
[Unicità di Prolog](#)



# 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  $3+5*2$



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

**Derivate**

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

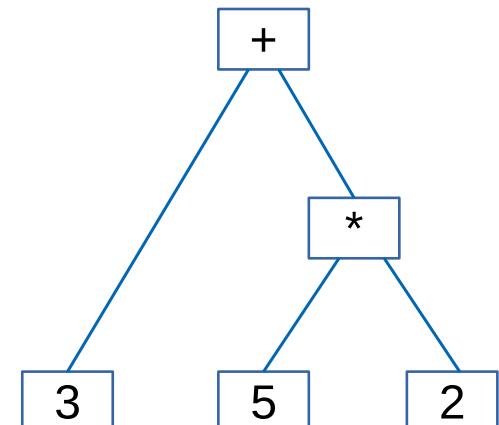
[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## ■ 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  $3+5*2$



## ■ Questo rende il calcolo simbolico particolarmente semplice. Facciamo un esempio basato sulle derivate, stile Matlab



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

**Derivate**

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
der(sen(X), X, cos(X)).
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

**Derivate**

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

**Derivate**

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
der(log(X), X, 1/X).
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
der(log(X), X, 1/X).
der(F+G, X, DF+DG) :-
 der(F, X, DF), der(G, X, DG).
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
der(log(X), X, 1/X).
der(F+G, X, DF+DG) :-
 der(F, X, DF), der(G, X, DG).
der(F-G, X, DF-DG) :-
 der(F, X, DF), der(G, X, DG).
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

**Derivate**

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
der(log(X), X, 1/X).
der(F+G, X, DF+DG) :-
 der(F, X, DF), der(G, X, DG).
der(F-G, X, DF-DG) :-
 der(F, X, DF), der(G, X, DG).
der(F*G, X, F*Dg+Df*G) :-
 der(F, X, DF), der(G, X, DG).
...
```



# Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

**Derivate**

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

**der(Expr,X,D)** è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
der(X^N, X, N*X^N1) :-
 N1 is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
der(log(X), X, 1/X).
der(F+G, X, DF+DG) :-
 der(F, X, DF), der(G, X, DG).
der(F-G, X, DF-DG) :-
 der(F, X, DF), der(G, X, DG).
der(F*G, X, F*Dg+Df*G) :-
 der(F, X, DF), der(G, X, DG).
...
```

```
?- der(x^3*cos(x), x, D).

D = x^3* -sen(x)+3*x^2*cos(x).
```

[Si possono aggiungere predicati per semplificare il risultato]



# Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## Usando `is` definire

1. un predicato `len(L)` che conta gli elementi della lista `L`



# Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## Usando `is` definire

1. un predicato `len(L)` che conta gli elementi della lista `L`
2. un predicato `fatt(N,F)` tale che `F` è il fattoriale di `N`



# Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## Usando `is` definire

1. un predicato `len(L)` che conta gli elementi della lista `L`
2. un predicato `fatt(N,F)` tale che `F` è il fattoriale di `N`
3. un predicato `sum(L,S)` tale che `S` è la somma degli elementi nella lista `L`



# Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[\*\*In che consiste\*\*](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

## ■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)



# Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

## ■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

## ■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```



# Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

## ■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

## ■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X



# Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

**In che consiste**

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

## ■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

## ■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X
- ◆ per ciascuno di essi verifica se è effettivamente una soluzione (test)



# Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

**In che consiste**

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

## ■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

## ■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X
- ◆ per ciascuno di essi verifica se è effettivamente una soluzione (test)
- ◆ se sì, restituisce la soluzione; altrimenti fa backtrack e torna a generare il successivo candidato



# Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

## ■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

## ■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X
- ◆ per ciascuno di essi verifica se è effettivamente una soluzione (test)
- ◆ se sì, restituisce la soluzione; altrimenti fa backtrack e torna a generare il successivo candidato
- ◆ se si chiedono altre risposte, genera altre soluzioni



# Confronto con gli altri paradigmi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo candidato;
while not test(X) and X ≠ null do
 X := prossimo_candidato(X)
return X
```



# Confronto con gli altri paradigmi

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo_candidato;
while not test(X) and X ≠ null do
 X := prossimo_candidato(X)
return X

/* paradigma funzionale */
fun soluzione(X) =
 if test(X) then X
 else soluzione(prossimo_candidato(X))
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)



# Confronto con gli altri paradigmi

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo_candidato;
while not test(X) and X ≠ null do
 X := prossimo_candidato(X)
return X

/* paradigma funzionale */
fun soluzione(X) =
 if test(X) then X
 else soluzione(prossimo_candidato(X))

/* invocare così */
soluzione(primo_candidato);
```

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog



# Confronto con gli altri paradigmi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo_candidato;
while not test(X) and X ≠ null do
 X := prossimo_candidato(X)
return X

/* paradigma funzionale */
fun soluzione(X) =
 if test(X) then X
 else soluzione(prossimo_candidato(X))

/* invocare così */
soluzione(primo_candidato);
```

- In entrambi i casi verrebbe generata solo la prima soluzione trovata (se si vogliono le altre occorre complicare il codice)
  - ◆ l'utilità del generare tutte le soluzioni sarà illustrata programmando l'AI di un gioco



# Esempi di programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

## ■ Ricerca dei membri pari di una lista

```
/* stile generate and test */
membro_pari(X,L) :- member(X,L), 0 is X mod 2.
```



# Esempi di programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

**In che consiste**

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

## ■ 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).
```



# Esempi di programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

## ■ 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



# Esempi di programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

**In che consiste**

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

## ■ 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: la query congiuntiva

```
member(X,Lista), predicato(X).
```

è l'analogo di: filter *predicato* Lista



# Applicazione ai giochi

## ■ Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-
 possible_move(Player ,Move), optimal(Player ,Move).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)



# Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-
 possible_move(Player ,Move), optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test



# Applicazione ai giochi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-
 possible_move(Player ,Move), optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test
- Nota: *optimal* mi dice se Move è la prima mossa di una strategia di gioco ottima → posso usare *optimal* per analizzare il gioco



# Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-
 possible_move(Player ,Move), optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test
- Nota: *optimal* mi dice se Move è la prima mossa di una strategia di gioco ottima → posso usare *optimal* per analizzare il gioco
  - ◆ ad es. il primo giocatore ha una strategia vincente?



# Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

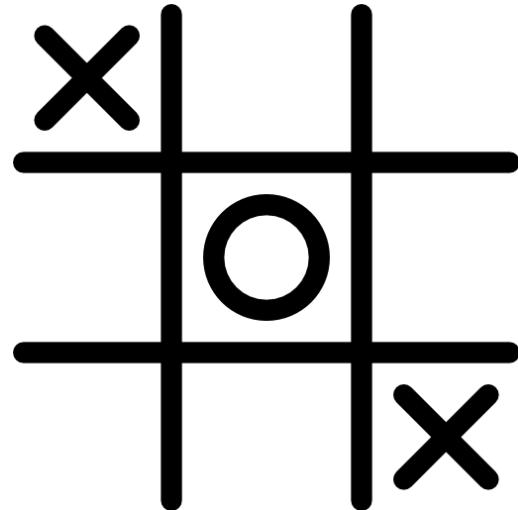
- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-
 possible_move(Player ,Move), optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test
- Nota: *optimal* mi dice se Move è la prima mossa di una strategia di gioco ottima → posso usare *optimal* per analizzare il gioco
  - ◆ ad es. il primo giocatore ha una strategia vincente?
  - ◆ mostrerà che è utile generare *tutte* le soluzioni



## Caso di studio: Tic tac toe (aka tris)



Un semplice programma di AI che gioca a tris (imbattibile)

- una pagina di codice per “l’intelligenza”
- una pagina per i turni e l’interfaccia utente

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

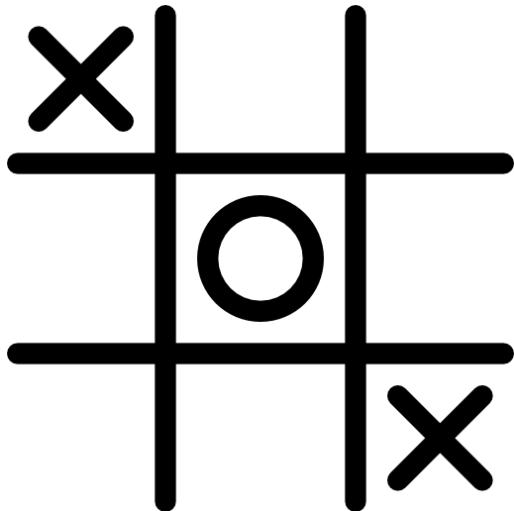
Analisi del gioco

GUI

[Unicità di Prolog](#)



## Caso di studio: Tic tac toe (aka tris)



Un semplice programma di AI che gioca a tris (imbattibile)

- una pagina di codice per “l’intelligenza”
- una pagina per i turni e l’interfaccia utente

- Effettua una ricerca della strategia ottima:
  - ◆ ne adotta una vincente, se esiste
  - ◆ altrimenti mira al pareggio

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

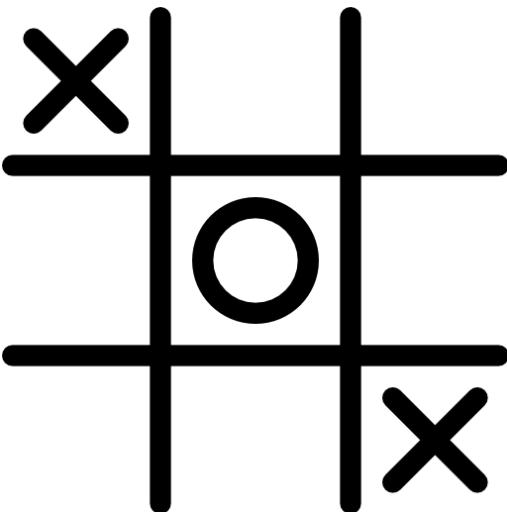
Analisi del gioco

GUI

[Unicità di Prolog](#)



# Caso di studio: Tic tac toe (aka tris)



Un semplice programma di AI che gioca a tris (imbattibile)

- una pagina di codice per “l’intelligenza”
- una pagina per i turni e l’interfaccia utente

- Effettua una ricerca della strategia ottima:
  - ◆ ne adotta una vincente, se esiste
  - ◆ altrimenti mira al pareggio
- Approccio *generate-and-test* (programmazione nondeterministica) alla scelta della mossa. Ad ogni turno:
  1. genera una mossa
  2. verifica se è ottimale (la prima di una strategia ottima)



# Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```



# Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```

```
adversary(x,o).
```

```
adversary(o,x).
```



# Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```

```
adversary(x,o).
```

```
adversary(o,x).
```

```
win(P, [[P,P,P],_,_]).
```

```
win(P, [_,[P,P,P],_]).
```

```
win(P, [_,_,[P,P,P]]).
```



# Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```

```
adversary(x,o).
```

```
adversary(o,x).
```

```
win(P, [[P,P,P],_,_]).
```

```
win(P, [_,[P,P,P],_]).
```

```
win(P, [_,_,[P,P,P]]).
```

```
win(P, [[P,_,_],[P,_,_],[P,_,_]]).
```

```
win(P, [[_,P,_],[_,P,_],[_,P,_]]).
```

```
win(P, [[_,_,P],[_,_,P],[_,_,P]])
```



# Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
start([[1,2,3],[4,5,6],[7,8,9]]).

adversary(x,o).
adversary(o,x).

win(P, [[P,P,P],_,_]).
win(P, [_,[P,P,P],_]).
win(P, [_,_,[P,P,P]]).
win(P, [[P,_,_],[P,_,_],[P,_,_]]).
win(P, [[_,P,_],[_,P,_],[_,P,_]]).
win(P, [[_,_,P],[_,_,P],[_,_,P]]).
win(P, [[P,_,_],[_,P,_],[_,_,P]]).
win(P, [[_,_,P],[_,P,_],[P,_,_]]).
```



# Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
start([[1,2,3],[4,5,6],[7,8,9]]).

adversary(x,o).
adversary(o,x).

win(P, [[P,P,P],_,_]).
win(P, [_,[P,P,P],_]).
win(P, [_,_,[P,P,P]]).
win(P, [[P,_,_],[P,_,_],[P,_,_]]).
win(P, [[_,P,_],[_,P,_],[_,P,_]]).
win(P, [[_,_,P],[_,_,P],[_,_,P]]).
win(P, [[P,_,_],[_,P,_],[_,_,P]]).
win(P, [[_,_,P],[_,P,_],[P,_,_]]).

non_final(Board) :-
 \+ win(_,Board),
 member(Row,Board),
 member(Cell,Row),
 number(Cell).
```



# Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
start([[1,2,3],[4,5,6],[7,8,9]]).

adversary(x,o).
adversary(o,x).

win(P, [[P,P,P],_,_]).
win(P, [_,[P,P,P],_]).
win(P, [_,_,[P,P,P]]).
win(P, [[P,_,_],[P,_,_],[P,_,_]]).
win(P, [[_,P,_],[_,P,_],[_,P,_]]).
win(P, [[_,_,P],[_,_,P],[_,_,P]]).
win(P, [[P,_,_],[_,P,_],[_,_,P]]).
win(P, [[_,_,P],[_,P,_],[P,_,_]]).

non_final(Board) :-
 \+ win(_,Board),
 member(Row,Board),
 member(Cell,Row),
 number(Cell).

final(Board) :-
 \+ non_final(Board).
```



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)`



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- **P** (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, `niente = IN-OUT`



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, niente = IN-OUT

```
move(P, N, Board1, Board2) :-
 \+ win(_, Board1),
```



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, niente = IN-OUT

```
move(P, N, Board1, Board2) :-
 \+ win(_, Board1),
 append(RowsBefore, [Row | RowsAfter], Board1),
 append(CellsBefore, [N | CellsAfter], Row),
```



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, niente = IN-OUT

```
move(P, N, Board1, Board2) :-
 \+ win(_, Board1),
 append(RowsBefore, [Row | RowsAfter], Board1),
 append(CellsBefore, [N | CellsAfter], Row),
 number(N),
```



# Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `move(+P,N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, niente = IN-OUT

```
move(P, N, Board1, Board2) :-
 \+ win(_, Board1),
 append(RowsBefore, [Row | RowsAfter], Board1),
 append(CellsBefore, [N | CellsAfter], Row),
 number(N),
 append(CellsBefore, [P | CellsAfter], NewRow),
 append(RowsBefore, [NewRow | RowsAfter], Board2).
```



# Tic tac toe – Le strategie (generate and test)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Schema: `has_XXX_strat(+P,-Move,+Board)` dove

- `P` (player) è il simbolo ‘x’ oppure ‘o’
- `Move` è la prima mossa della strategia
- `Board` è l’attuale scacchiera

Significato: *nella situazione descritta da Board, P ha una strategia di tipo XXX (vincente o non-perdente) che inizia con Move*



# Tic tac toe – Le strategie (generate and test)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[In che consiste](#)

**Tic tac toe**

[I/O elementare](#)

[Analisi del gioco](#)

[GUI](#)

[Unicità di Prolog](#)

Schema: `has_XXX_strat(+P,-Move,+Board)` dove

- `P` (player) è il simbolo ‘x’ oppure ‘o’
- `Move` è la prima mossa della strategia
- `Board` è l’attuale scacchiera

Significato: *nella situazione descritta da Board, P ha una strategia di tipo XXX (vincente o non-perdente) che inizia con Move*

```
has_win_strat(P,_,Board) :-
 win(P,Board).
has_win_strat(P,Move,Board) :-
 move(P,Move,Board,Board2),
 adversary(P,Adv),
 \+ has_tie_strat(Adv,_,Board2).
```



# Tic tac toe – Le strategie (generate and test)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

**Tic tac toe**

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

Schema: `has_XXX_strat(+P,-Move,+Board)` dove

- `P` (player) è il simbolo ‘x’ oppure ‘o’
- `Move` è la prima mossa della strategia
- `Board` è l’attuale scacchiera

Significato: *nella situazione descritta da Board, P ha una strategia di tipo XXX (vincente o non-perdente) che inizia con Move*

```
has_win_strat(P,_,Board) :-
 win(P,Board).
has_win_strat(P,Move,Board) :-
 move(P,Move,Board,Board2),
 adversary(P,Adv),
 \+ has_tie_strat(Adv,_,Board2).

has_tie_strat(_,_,Board) :-
 final(Board),
 \+ win(_,Board).
has_tie_strat(P,Move,Board) :-
 move(P,Move,Board,Board2),
 adversary(P,Adv),
 \+ has_win_strat(Adv,_,Board2).
```



# Tic tac toe – L'interfaccia utente testuale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

(per le interfacce grafiche vedere l'estensione XPCE,  
<https://www.swi-prolog.org/packages/xpce/>)

```
cpu move: 1
x|2|3
4|5|6
7|8|9

Player o insert your move [1-9]: 8
x|2|3
4|5|6
7|o|9
```



# Tic tac toe – L'interfaccia utente testuale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

(per le interfacce grafiche vedere l'estensione XPCE,  
<https://www.swi-prolog.org/packages/xpce/>)

```
cpu move: 1
x|2|3
4|5|6
7|8|9

Player o insert your move [1-9]: 8
x|2|3
4|5|6
7|o|9
```

```
print_board([]).
print_board([Row|Rest]) :-
 format('`~a|`~a|`~a\n', Row),
 print_board(Rest).
```



# Tic tac toe – L'interfaccia utente testuale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

(per le interfacce grafiche vedere l'estensione XPCE,  
<https://www.swi-prolog.org/packages/xpce/>)

```
cpu move: 1
x|2|3
4|5|6
7|8|9

Player o insert your move [1-9]: 8
x|2|3
4|5|6
7|o|9
```

```
print_board([]).
print_board([Row|Rest]) :-
 format('`~a|`~a|`~a \n', Row),
 print_board(Rest).

read_move(Player,Move) :-
 format('\nPlayer ~a insert your move [1-9]: ', [Player]),
 get_single_char(Char), put_char(Char), nl,
 Move is Char-48,
 Move >= 1, Move =< 9.
```



# Tic tac toe – I turni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
turn(_,_,Board) :-
 final(Board),
 \+ win(_,Board),
 format('`The game ends in a tie.\n').
```



# Tic tac toe – I turni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

```
turn(_,_,Board) :-
 final(Board),
 \+ win(_,Board),
 format('`\nThe game ends in a tie.\n').
```

```
turn(P,_,Board) :-
 win(_,Board),
 member(Adv,[user,cpu]), Adv \= P,
 format('`\nThe ~a wins! \n', [Adv]).
```



# Tic tac toe – I turni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

```
turn(_,_,Board) :-
 final(Board),
 \+ win(_,Board),
 format('`\nThe game ends in a tie.\n').

turn(P,_,Board) :-
 win(_,Board),
 member(Adv,[user,cpu]), Adv \= P,
 format('`\nThe ~a wins! \n', [Adv]).

turn(user,P,Board) :-
 read_move(P,M),
 move(P, M, Board, Board2),
 print_board(Board2),
 adversary(P,Adv),
 turn(cpu,Adv,Board2).
```



# Tic tac toe – I turni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

```
turn(_,_,Board) :-
 final(Board),
 \+ win(_,Board),
 format('`\nThe game ends in a tie.\n').

turn(P,_,Board) :-
 win(_,Board),
 member(Adv,[user,cpu]), Adv \= P,
 format('`\nThe ~a wins! \n', [Adv]).

turn(user,P,Board) :-
 read_move(P,M),
 move(P, M, Board, Board2),
 print_board(Board2),
 adversary(P,Adv),
 turn(cpu,Adv,Board2).

turn(cpu,P,Board) :-
 (has_win_strat(P,Move,Board); has_tie_strat(P,Move,Board)),
 format('`ncpu move: ~a \n', [Move]),
 move(P, Move, Board, Board2),
 print_board(Board2),
 adversary(P,Adv),
 turn(user,Adv,Board2).
```



## Tic-tac-toe – II “main”

Due modalità di esecuzione:

- Specificando liberamente chi inizia e che simbolo usa

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)



## Tic-tac-toe – II “main”

Due modalità di esecuzione:

- Specificando liberamente chi inizia e che simbolo usa

```
go(First ,Symbol) :-
 member(First ,[user ,cpu]) ,
 start(Board) ,
 turn(First ,Symbol ,Board).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)



# Tic-tac-toe – II “main”

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

[Unicità di Prolog](#)

Due modalità di esecuzione:

■ Specificando liberamente chi inizia e che simbolo usa

```
go(First ,Symbol) :-
 member(First ,[user ,cpu]) ,
 start(Board) ,
 turn(First ,Symbol ,Board).
```

Esempio di invocazione

```
?- go(user ,x).
```



# Tic-tac-toe – II “main”

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

Due modalità di esecuzione:

- Specificando liberamente chi inizia e che simbolo usa

```
go(First ,Symbol) :-
 member(First ,[user ,cpu]) ,
 start(Board) ,
 turn(First ,Symbol ,Board).
```

Esempio di invocazione

```
?- go(user ,x).
```

- Estrazione a sorte di chi inizia

```
main :-
 Choice is random(2) ,
 nth0(Choice ,[(user ,o),(cpu ,x)] ,(First ,Symbol)) ,
 format('`nThe ~a starts `n' , [First]) ,
 start(Board) ,
 turn(First ,Symbol ,Board) ,
 halt.
```

Demo



# Tic-tac-toe – Analisi del gioco

- Esiste una strategia vincente per il primo giocatore?

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

[Unicità di Prolog](#)



# Tic-tac-toe – Analisi del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

[Unicità di Prolog](#)

## ■ Esiste una strategia vincente per il primo giocatore?

```
?- start(Board), has_win_strat(x, Move, Board).
false.
```

NO: nessuna mossa iniziale garantisce al primo giocatore di vincere



# Tic-tac-toe – Analisi del gioco

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

- Esiste una strategia vincente per il primo giocatore?

```
?- start(Board), has_win_strat(x, Move, Board).
false.
```

NO: nessuna mossa iniziale garantisce al primo giocatore di vincere

- Esiste una strategia vincente per il secondo giocatore?



# Tic-tac-toe – Analisi del gioco

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

Unicità di Prolog

## ■ Esiste una strategia vincente per il primo giocatore?

```
?- start(Board), has_win_strat(x, Move, Board).
false.
```

NO: nessuna mossa iniziale garantisce al primo giocatore di vincere

## ■ Esiste una strategia vincente per il secondo giocatore?

```
?- start(B0), has_tie_strat(x, Mv, B0).
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
Mv = 1 ;
...
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
Mv = 9 ;
false.
```

NO: ogni mossa iniziale permette al primo giocatore di pareggiare (se non commette errori)

- ◆ tutte le mosse iniziali fanno parte di una strategia di pareggio



# Tic-tac-toe – Analisi del gioco (II)

- Come può pareggiare il secondo giocatore?

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

[Unicità di Prolog](#)



# Tic-tac-toe – Analisi del gioco (II)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

## ■ Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,1,B0,B1), has_tie_strat(o,Mv,B1).
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
B1 = [[x, 2, 3], [4, 5, 6], [7, 8, 9]],
Mv = 5 ;
false.
```

Se la prima mossa è su un angolo, l'unica risposta è 5, in tutti gli altri casi vince il primo giocatore



# Tic-tac-toe – Analisi del gioco (II)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

## ■ Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,1,B0,B1), has_tie_strat(o,Mv,B1).
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
B1 = [[x, 2, 3], [4, 5, 6], [7, 8, 9]],
Mv = 5 ;
false.
```

Se la prima mossa è su un angolo, l'unica risposta è 5, in tutti gli altri casi vince il primo giocatore

```
?- start(B0), move(x,2,B0,B1), has_tie_strat(o,Mv,B1).
...
Mv = 1 ;
...
Mv = 3 ;
...
Mv = 5 ;
...
Mv = 8 ;
false.
```

Quindi se la prima mossa è 2, meglio non rispondere con 4, 6, 7 o 9!



# Tic-tac-toe – Analisi del gioco (III)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

[Unicità di Prolog](#)

## ■ Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,5,B0,B1), has_tie_strat(o,Mv,B1).
```



# Tic-tac-toe – Analisi del gioco (III)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione  
nondeterministica

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

GUI

Unicità di Prolog

- Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,5,B0,B1), has_tie_strat(o,Mv,B1).
...
Mv = 1 ;
...
Mv = 3 ;
...
Mv = 7 ;
...
Mv = 9 ;
false.
```

Se la prima mossa è 5, si deve scegliere una casella d'angolo

- Tutte queste affermazioni si possono verificare empiricamente forzando il primo passo:

```
?- start(B0), move(x,5,B0,B1), turn(user,o,B1).
```



# Tic-tac-toe – Compilazione in codice stand-alone

```
swipl --goal=main --stand_alone=true -o tic-tac-toe
 -c tic-tac-toe.pl
```

## ■ Spiegazione delle opzioni:

- ◆ **--stand\_alone**: crea un codice direttamente eseguibile  
(invece di far partire la macchina virtuale)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

[Unicità di Prolog](#)



# Tic-tac-toe – Compilazione in codice stand-alone

```
swipl --goal=main --stand_alone=true -o tic-tac-toe
 -c tic-tac-toe.pl
```

## ■ Spiegazione delle opzioni:

- ◆ --stand\_alone: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)
- ◆ -o: nome del file oggetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

[Unicità di Prolog](#)



# Tic-tac-toe – Compilazione in codice stand-alone

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

**Analisi del gioco**

GUI

[Unicità di Prolog](#)

```
swipl --goal=main --stand_alone=true -o tic-tac-toe
 -c tic-tac-toe.pl
```

## ■ Spiegazione delle opzioni:

- ◆ --stand\_alone: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)
- ◆ -o: nome del file oggetto
- ◆ -c: nome del file sorgente



# Tic-tac-toe – Compilazione in codice stand-alone

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

[Analisi del gioco](#)

GUI

[Unicità di Prolog](#)

```
swipl --goal=main --stand_alone=true -o tic-tac-toe
 -c tic-tac-toe.pl
```

## ■ Spiegazione delle opzioni:

- ◆ --stand\_alone: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)
- ◆ -o: nome del file oggetto
- ◆ -c: nome del file sorgente
- ◆ --goal: il goal da invocare quando il file viene eseguito



# Cenni a una possibile interfaccia grafica con XPCE





# Cenni a una possibile interfaccia grafica con XPCE



```
gui :-
 /* costruzione buttoni */

 new(Msg,button('Your turn')) ,
 new(B1,button('')) ,
 ...
 new(B9,button('')) ,
```



# Cenni a una possibile interfaccia grafica con XPCE



```
gui :-
 /* costruzione buttoni */

 new(Msg,button('Your turn')),
 new(B1,button('')),
 ...
 new(B9,button('')),
```

```
/* costruzione finestra */
new(Dialog, dialog('Tic Tac Toe')),
send(Dialog, gap, size(10,10)),
send(Dialog, append, B1),
send(Dialog, append, B2, right),
send(Dialog, append, B3, right),
send(Dialog, append, B4, below),
...
send(Dialog, append, Msg, below),
send(Msg, alignment, left),
send(Dialog, open),
/* il gioco inizia */
gui_go(Msg,Buttons).
```



# Cenni a una possibile interfaccia grafica con XPCE



```
gui :-
 /* costruzione button */

 new(Msg,button('Your turn')),
 new(B1,button('')),
 ...
 new(B9,button('')),
```

```
/* costruzione finestra */
new(Dialog, dialog('Tic Tac Toe')),
send(Dialog, gap, size(10,10)),
send(Dialog, append, B1),
send(Dialog, append, B2, right),
send(Dialog, append, B3, right),
send(Dialog, append, B4, below),
...
send(Dialog, append, Msg, below),
send(Msg, alignment, left),
send(Dialog, open),
/* il gioco inizia */
gui_go(Msg,Buttons).
```

```
/* inizializzazione button */
send(Bi, message,
 message(@prolog,click,I,B1,B2,...)),
send(Bi, label, ''),
send(Bi, size, size(45,45))
...

/* quando si clicca */
send(Bi, message,
 message(@prolog,true),
send(Bi, label, 'X')),
...
```



# Compilazione stand-alone della versione grafica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

In che consiste

Tic tac toe

I/O elementare

Analisi del gioco

**GUI**

[Unicità di Prolog](#)

```
swipl --goal=gui --stand_alone=true --pce -o tic-tac-toe-gui
 -c tic-tac-toe-gui.pl
```

## ■ Spiegazione delle opzioni:

- ◆ --pce rende disponibili le primitive per la GUI
- ◆ le altre sono come nell'esempio precedente

## ■ Attenzione! Se si usa '@' bisogna dichiararlo come operatore unario prefisso (altrimenti viene generato un syntax error)

```
/* inserire all'inizio del programma */
:- op(1, fx, @).
```



# Riassumendo: caratteristiche uniche di Prolog

## ■ 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

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)



# Riassumendo: caratteristiche uniche di Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## ■ 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

## ■ Programmazione nondeterministica

- ◆ con ricerca automatica delle soluzioni
- ◆ basato sul backtracking (cioè il meccanismo di visita dei search tree)



# Riassumendo: caratteristiche uniche di Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione  
nondeterministica](#)

[Unicità di Prolog](#)

## ■ 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

## ■ Programmazione nondeterministica

- ◆ con ricerca automatica delle soluzioni
- ◆ basato sul backtracking (cioè il meccanismo di visita dei search tree)

## ■ Restano molti aspetti interessanti che purtroppo non abbiamo il tempo di illustrare

- ◆ strutture dati parziali (permettono append e creazione dizionari in tempo *costante*)
- ◆ meta-predicati / reflection
- ◆ aggregati (setof)
- ◆ forall ...

---

# Linguaggi di Programmazione I – Lezione 4

Prof. Marcello Sette  
mailto://marcello.sette@gmail.com  
<http://sette.dnsalias.org>

18 marzo 2010



# Panoramica della lezione

Parametrizzazione di procedure

Bibliografia



## Parametrizzazione di procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

# Parametrizzazione di procedure



# Parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.



# Parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.
- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.



# Parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.
- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Si possono distinguere:
  - ◆ parametri IN: sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;



# Parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.
- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Si possono distinguere:
  - ◆ parametri IN: sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;
  - ◆ parametri OUT: sono passati dall'unità chiamata alla unità chiamante al momento della terminazione della prima;



# Parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.
- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Si possono distinguere:
  - ◆ parametri IN: sono passati dalla unità chiamante alla 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: servono a far transitare le informazioni in entrambe le direzioni.



# Parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.
- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Si possono distinguere:
  - ◆ parametri IN: sono passati dalla unità chiamante alla 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: servono a far transitare le informazioni in entrambe le direzioni.
- Devono essere specificati in due punti:



# Parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.
- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Si possono distinguere:
  - ◆ parametri IN: sono passati dalla unità chiamante alla 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: servono a far transitare le informazioni in entrambe le direzioni.
- Devono essere specificati in due punti:
  - ◆ nella definizione della procedura: *parametri formali*;



# Parametri

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

- Sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura.
- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Si possono distinguere:
  - ◆ parametri IN: sono passati dalla unità chiamante alla 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: servono a far transitare le informazioni in entrambe le direzioni.
- Devono essere specificati in due punti:
  - ◆ nella definizione della procedura: *parametri formali*;
  - ◆ nelle invocazioni della procedura: *parametri attuali*.



# Associazione dei parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- **Regola comune:** nella definizione deve essere specificato il tipo dei parametri formali; nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.



# Associazione dei parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

■ **Regola comune:** nella definizione deve essere specificato il tipo dei parametri formali; nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.

■ **Eccezioni comuni:**



# Associazione dei parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

■ **Regola comune:** nella definizione deve essere specificato il tipo dei parametri formali; nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.

■ **Eccezioni comuni:**

- ◆ 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).



# Associazione dei parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

■ **Regola comune:** nella definizione deve essere specificato il tipo dei parametri formali; nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.

■ **Eccezioni comuni:**

- ◆ 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).
- ◆ permettere come eccezione solo quella degli array a dimensione variabile; il legame di tipo (e l'eventuale controllo di consistenza) verrà realizzato durante l'esecuzione.



# Associazione dei parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

■ **Regola comune:** nella definizione deve essere specificato il tipo dei parametri formali; nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.

■ **Eccezioni comuni:**

- ◆ 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).
- ◆ permettere come eccezione solo quella degli 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;



# Associazione dei parametri

Parametrizzazione di  
procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

■ **Regola comune:** nella definizione deve essere specificato il tipo dei parametri formali; nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.

■ **Eccezioni comuni:**

- ◆ 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).
- ◆ permettere come eccezione solo quella degli 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;



## Esempio

Parametrizzazione di procedure

Parametri

Associazione dei ...

**Esempio**

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Data l'intestazione della seguente procedura (ADA):

```
procedure TEST (A: in Atyp; b: in out Btype; C: out Ctype)
```



## Esempio

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Data l'intestazione della seguente procedura (ADA):

```
procedure TEST (A: in Atyp; b: in out Btyp; C: out Ctyp)
```

allora una invocazione che usi associazione per posizione è:

```
TEST(X, Y, Z);
```



## Esempio

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Data l'intestazione della seguente procedura (ADA):

```
procedure TEST (A: in Atyp; b: in out Btyp; C: out Ctyp)
```

allora una 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);
```



## Esempio

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Data l'intestazione della seguente procedura (ADA):

```
procedure TEST (A: in Atyp; b: in out Btyp; C: out Ctyp)
```

allora una 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 ulteriore tecnica è la cosiddetta *associazione di default*. Essa permette di specificare valori di default ai parametri formali che non sono stati legati a valori da parametri attuali.



## Parametri IN

Possono essere realizzati in due modi:

Parametrizzazione di procedure

Parametri

Associazione dei . . .

Esempio

**Parametri IN**

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia



## Parametri IN

Possono essere realizzati in due modi:

1. con un riferimento;

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

**Parametri IN**

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia



## Parametri IN

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei ...

Esempio

**Parametri IN**

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

[Bibliografia](#)

---

Possono essere realizzati in due modi:

1. con un riferimento; in questo caso la locazione del parametro attuale diventa la locazione del parametro formale;



# Parametri IN

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei ...

Esempio

**Parametri IN**

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

[Bibliografia](#)

---

Possono essere realizzati in due modi:

1. con un riferimento; in questo caso la locazione del parametro attuale diventa la locazione del parametro formale;  
poiché il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura;



## Parametri IN

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. con un riferimento; in questo caso la locazione del parametro attuale diventa la locazione del parametro formale;  
poiché il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura;
2. con una copia;



## Parametri IN

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. con un riferimento; in questo caso la locazione del parametro attuale diventa la locazione del parametro formale;  
poiché il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura;
2. con una copia; in questo caso in una nuova locazione, quella del parametro formale, viene copiato il valore del parametro attuale;



## Parametri IN

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. con un riferimento; in questo caso la locazione del parametro attuale diventa la locazione del parametro formale;  
poiché il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura;
2. con una copia; in questo caso in una nuova locazione, quella del parametro formale, viene copiato il valore del parametro attuale; parametro formale visto come variabile locale;



## Parametri IN

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. con un riferimento; in questo caso la locazione del parametro attuale diventa la locazione del parametro formale;  
poiché il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura;
2. con una copia; in questo caso in una nuova locazione, quella del parametro formale, viene copiato il valore del parametro attuale; parametro formale visto come variabile locale;  
modifica permessa, perché valida solo nell'ambiente di esecuzione della procedura.



## Parametri IN

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. con un riferimento; in questo caso la locazione del parametro attuale diventa la locazione del parametro formale;  
poiché il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura;
2. con una copia; in questo caso in una nuova locazione, quella del parametro formale, viene copiato il valore del parametro attuale; parametro formale visto come variabile locale;  
modifica permessa, perché valida solo nell'ambiente di esecuzione 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.



# Parametri OUT

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei . . .

Esempio

Parametri IN

**Parametri OUT**

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

[Bibliografia](#)

---

Possono essere realizzati:



# Parametri OUT

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

**Parametri OUT**

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati:

1. con un riferimento;



# Parametri OUT

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei ...

Esempio

Parametri IN

**Parametri OUT**

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

[Bibliografia](#)

---

Possono essere realizzati:

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



# Parametri OUT

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati:

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

Non esistono regole generali nemmeno tra diverse versioni di uno stesso linguaggio

- Ada 83 proibisce di “leggere” i parametri OUT
- Le versioni successive invece lo permettono



# Parametri IN OUT

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

**Parametri IN OUT**

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

[Bibliografia](#)

---

Sono la combinazione dei due precedenti.



## Parametri IN OUT

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

**Parametri IN OUT**

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

[Bibliografia](#)

---

Sono la combinazione dei due precedenti. Anch'essi possono essere realizzati:

1. con un riferimento; non ci sono limitazioni all'uso all'interno della procedura;



## Parametri IN OUT

[Parametrizzazione di procedure](#)

---

[Parametri](#)

[Associazione dei ...](#)

[Esempio](#)

[Parametri IN](#)

[Parametri OUT](#)

**Parametri IN OUT**

[Aliasing](#)

[Procedure come ...](#)

[Macro](#)

[Esercizio](#)

[Funzioni](#)

[Bibliografia](#)

---

Sono la combinazione dei due precedenti. Anch'essi possono essere realizzati:

1. con un riferimento; non ci sono limitazioni all'uso all'interno della procedura;
2. con una copia; avvengono due processi di copia, uno durante l'attivazione ed uno durante la terminazione della procedura.



# Aliasing

È la possibilità di riferirsi alla stessa locazione con nomi diversi.

Parametrizzazione di procedure

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

**Aliasing**

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia



# Aliasing

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

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

**Aliasing**

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia



# Aliasing

Parametrizzazione di procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

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

```
program MAIN;
 var
 A: integer;
 procedure TEST (var X, Y: integer);
 begin
 X := A + Y;
 writeln(A, X, Y)
 end;
begin
 A := 1;
 TEST(A, A)
end.
```

Esercizio: determinare l'uscita del programma nel caso in cui i parametri VAR siano realizzati *per riferimento* e nel caso in cui siano realizzati *per copia*.



# Procedure come parametri di procedura

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

**Procedure come . . .**

Macro

Esercizio

Funzioni

[Bibliografia](#)

---

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



# Procedure come parametri di procedura

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

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

```
program MAIN;
 VAR a: real;
 procedure TESTPOS (X: real; procedure ERROR (MSG: string));
 begin
 if X <= 0 then ERROR ('Negative X in TESTPOS')
 end;
 procedure E1 (M: string);
 begin
 writeln('E1 error: ', M)
 end;
 procedure E2 (M: string);
 begin
 writeln('E2 error: ', M)
 end;
begin
 readln (A);
 TESTPOS(A, E1);
 TESTPOS(A, E2)
end.
```



## 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.

[Parametrizzazione di procedure](#)

---

[Parametri](#)

[Associazione dei ...](#)

[Esempio](#)

[Parametri IN](#)

[Parametri OUT](#)

[Parametri IN OUT](#)

[Aliasing](#)

[Procedure come ...](#)

**Macro**

[Esercizio](#)

[Funzioni](#)

[Bibliografia](#)

---



## Macro

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

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

Esempio: data la procedura

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

allora la chiamata `swap(x, y)` esegue il seguente brano di codice:

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



## Esercizio

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

**Esercizio**

Funzioni

Bibliografia

Determinare i problemi che nascono dai due programmi, se swap è una macro:

```
program main;
var
 i: integer;
 m: array [1..100] of integer;
 ...
begin
 ...
 swap(i, m[i]);
 ...
end.
```



## Esercizio

Parametrizzazione di  
procedure

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

**Esercizio**

Funzioni

Bibliografia

Determinare i problemi che nascono dai due programmi, se swap è una macro:

```
program main;
var
 i: integer;
 m: array [1..100] of integer;
 ...
begin
 ...
 swap(i, m[i]);
 ...
end.
```

```
program main;
var
 i, temp: integer;
 ...
begin
 ...
 swap(i, temp);
 ...
end.
```



## Funzioni

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei . . .

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come . . .

Macro

Esercizio

**Funzioni**

[Bibliografia](#)

---

Sono procedure che restituiscono un valore alla procedura chiamante.



## Funzioni

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

**Funzioni**

---

Bibliografia

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

- o creando una pseudovariabile nell'ambiente locale della procedura chiamata. Tale variabile può essere solo modificata; non è possibile l'accesso in lettura.



## Funzioni

[Parametrizzazione di procedure](#)

---

Parametri

Associazione dei ...

Esempio

Parametri IN

Parametri OUT

Parametri IN OUT

Aliasing

Procedure come ...

Macro

Esercizio

**Funzioni**

---

[Bibliografia](#)

---

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

- o creando una pseudovariabile nell'ambiente locale della procedura chiamata. Tale variabile può essere solo modificata; non è possibile l'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.



## Bibliografia

Parametrizzazione di  
procedure

Bibliografia

Bibliografia

- H. L. Dershem. M. J. Jipping. *Programming languages: structures and models*. Second edition. Cap. 5, par. 5.4.

# **Polimorfismo parametrico vs polimorfismo per inclusione**

# Esercizio

- Definire il tipo di dato “Stack” con operazioni
  - Push( element )
  - Pop()
  - Non “forzare” una specifica implementazione
  - Non “forzare” un tipo specifico per gli elementi
    - È una libreria: non sapete come la useranno
- Definire l'implementazione a lista concatenata:
  - LinkedStack
- Usate al meglio i costrutti di Java!

# Una soluzione

- Prima parte

```
public interface ObjectStack {
 public void push(Object el);
 public Object pop() throws EmptyStackException;
}
```

```
public class EmptyStackException extends Exception {
 static final long serialVersionUID = 99999;
}
```

# Una soluzione

## ■ Seconda parte

```
public class LinkedObjectStack implements ObjectStack {
 private class StackRecord {
 Object el;
 StackRecord next;
 StackRecord(Object el, StackRecord next){ this.el = el; this.next = next; }
 }
 StackRecord top;

 public void push(Object el){ top = new StackRecord(el, top); }

 public Object pop() throws EmptyStackException {
 Object res;
 if(top == null) throw new EmptyStackException();
 res = top.el;
 top = top.next;
 return res; }
}
```

# Una soluzione

## ■ Esempio di uso #1

```
ObjectStack so = new LinkedObjectStack();

so.push(1); // conversione implicita a Integer (autoboxing)
so.push(2); // e upcast automatico a Object
so.push(3);
```

```
try{
 while(true){
 System.out.println(((Integer)so.pop()).intValue());
 }
} catch(Exception e) { System.out.println("fine"); }
```

Se non faccio il downcast...

non posso usare questo metodo  
(specifico di Integer)

3  
2  
1  
fine

output

# Una soluzione

## ■ Esempio di uso #2

```
ObjectStack so = new LinkedObjectStack();

so.push(1); // conversione implicita a Integer (autoboxing)
so.push(2); // e upcast automatico a Object
so.push("A"); ←

try{
 while(true){
 System.out.println(((Integer)so.pop()).intValue());
 }
 catch(Exception e) { System.out.println("fine"); }
```

Se mi distraggo compila ancora ma...

output

```
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
```

# Una soluzione migliore: template

- Prima parte

```
public interface GenericStack<ElemType> {
 public void push(ElemType el);
 public ElemType pop() throws EmptyStackException;
}
```

# Una soluzione migliore: template

## ■ Seconda parte

```
public class GenericLinkedList<ElemType> implements GenericStack<ElemType> {
 private class StackRecord<EIType> {
 EIType el;
 StackRecord<EIType> next;
 StackRecord(EIType el, StackRecord<EIType> next){ this.el = el; this.next = next; }
 }
 StackRecord<ElemType> top;

 public void push(ElemType el) { top = new StackRecord<ElemType>(el, top); }
 public ElemType pop() throws EmptyStackException {
 ElemType res;
 if(top == null) throw new EmptyStackException();
 res = top.el;
 top = top.next;
 return res;
 }
}
```

# Una soluzione migliore: template

## ■ Esempio di uso #1

```
GenericStack< Integer > si = new GenericLinkedStack< Integer >();

si.push(1);
si.push(2);
si.push(3);
try{
 while(true){ System.out.println(si.pop().intValue()); }
}
catch(Exception e) { System.out.println("fine"); }
```

Non serve downcast:  
pop() restituisce Integer

3  
2  
1  
fine

output

# Una soluzione migliore: template

## ■ Esempio di uso #2

```
GenericStack< Integer > si = new GenericLinkedStack< Integer >();
```

```
si.push(1);
si.push(2);
si.push("A"); ←
try{
 while(true){ System.out.println(si.pop().intValue()); }
}
catch(Exception e) { System.out.println("fine"); }
```

push( Integer )

1. ERROR in mylists/Test2.java (at line 8)

```
si.push("A");
^^^^
```

output del compilatore!

The method push(Integer) in the type GenericStack<Integer> is not applicable  
for the arguments (String)

# Confronto

- Pro del polimorfismo parametrico:
  - Non richiede controlli di tipo a run time
  - Anticipa scoperta errori di tipo a tempo di compilazione
- Pro del polimorfismo per inclusione
  - Permette strutture dati eterogenee
  - Ad esempio, Stack di elementi di tipo diverso

# Prendere il meglio

- In molti casi
  - Servono collezioni di elementi eterogenei
  - Ma vogliamo usarli allo stesso modo
- Esempio:
  - Una scena è una *lista* di forme eterogenee (rettangoli, ellissi, linee, ecc.)
  - Vogliamo usarla per implementare *refresh*, che deve solo inviare un messaggio *draw()* a tutte le forme della lista
- Quindi
  - Identificare le modalità d'uso degli elementi
  - Scegliere una superclasse comune o fattorizzarle in una interfaccia
  - Usare la superclasse/interfaccia come argomento del template

# Note sull'implementazione

- In Java – internamente - sarebbe comunque uno stack di Object
  - Ma coi template dichiariamo che vogliamo metterci solo oggetti di un certo tipo
  - Per questo non possiamo legare il parametro a tipi elementari come int o float ma dobbiamo usare i wrapper
- In C++ ogni uso dei template fa compilare una nuova classe
  - Sorta di macro
  - Il compilatore inserisce nel programma la definizione di classe specializzata e la ricompila per ogni parametro attuale

---

# Linguaggi di Programmazione I – Lezione 3

Prof. Marcello Sette  
mailto://marcello.sette@gmail.com  
<http://sette.dnsalias.org>

16 marzo 2010



# Panoramica della lezione

Procedure come astrazioni

Record di attivazione

Propagazione dei data object

Bibliografia



Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di . . .

Ambiente di . . .

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

## Procedure come astrazioni



## Procedure

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di . . .

Ambiente di . . .

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

**Procedure** sono astrazioni di parti di programma in unità di esecuzione più piccole, come enunciati o espressioni, in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.



## Procedure

Procedure come astrazioni

Procedure

Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

**Procedure** sono astrazioni di parti di programma in unità di esecuzione più piccole, come enunciati o espressioni, in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.



## Procedure

Procedure come astrazioni

Procedure  
Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

**Procedure** sono astrazioni di parti di programma in unità di esecuzione più piccole, come enunciati o espressioni, in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.
- Unità di programmi indipendenti o con dipendenze ben specificate a livello più alto.



## Procedure

Procedure come astrazioni

Procedure

Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

**Procedure** sono astrazioni di parti di programma in unità di esecuzione più piccole, come enunciati o espressioni, in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.
- Unità di programmi indipendenti o con dipendenze ben specificate a livello più alto.
- Riusabilità di brani di programmi; riduzione errori.



# Astrazione procedurale

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di . . .

Ambiente di . . .

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

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



# Astrazione procedurale

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

- 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 di esecuzione attraverso un'altra unità più semplice.



# Astrazione procedurale

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

- 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 di esecuzione attraverso un'altra unità più semplice.
- In pratica è la rappresentazione di un blocco attraverso un enunciato o una espressione.



# Dichiarazione di procedura

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

- Causa la generazione di un oggetto analogo al Type Object (anche se in questo corso non esaminato).



# Dichiarazione di procedura

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

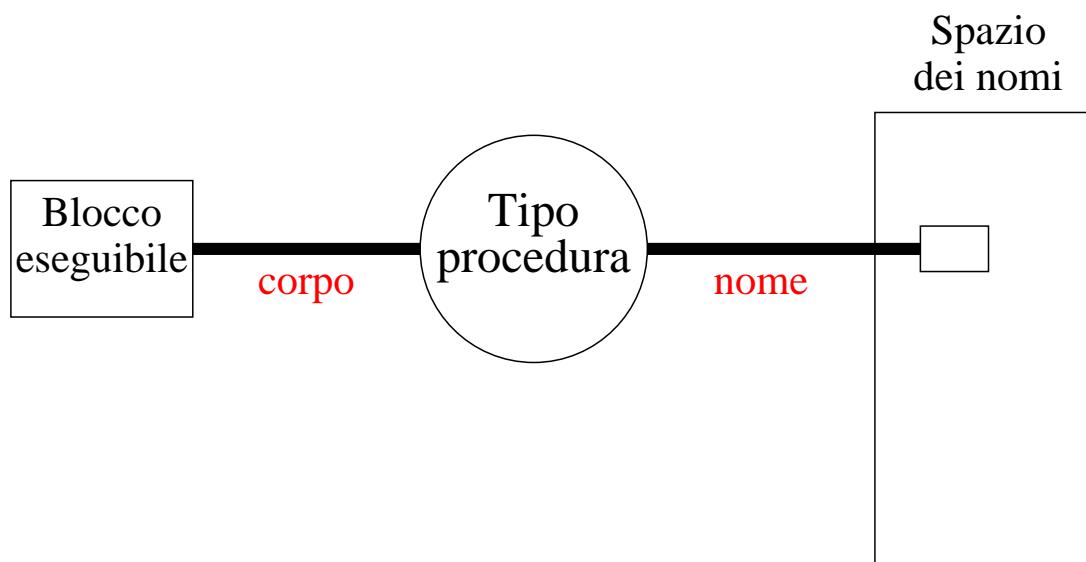
Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

- Causa la generazione di un oggetto analogo al Type Object (anche se in questo corso non esaminato).
- Il processo avviene durante la compilazione.





## Invocazione di una procedura

- Causa la generazione di un oggetto analogo al Data Object.

Procedure come astrazioni

Procedure  
Astrazione  
procedurale  
Dichiarazione

**Invocazione di . . .**

Ambiente di . . .

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia



# Invocazione di una procedura

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di . . .

Ambiente di . . .

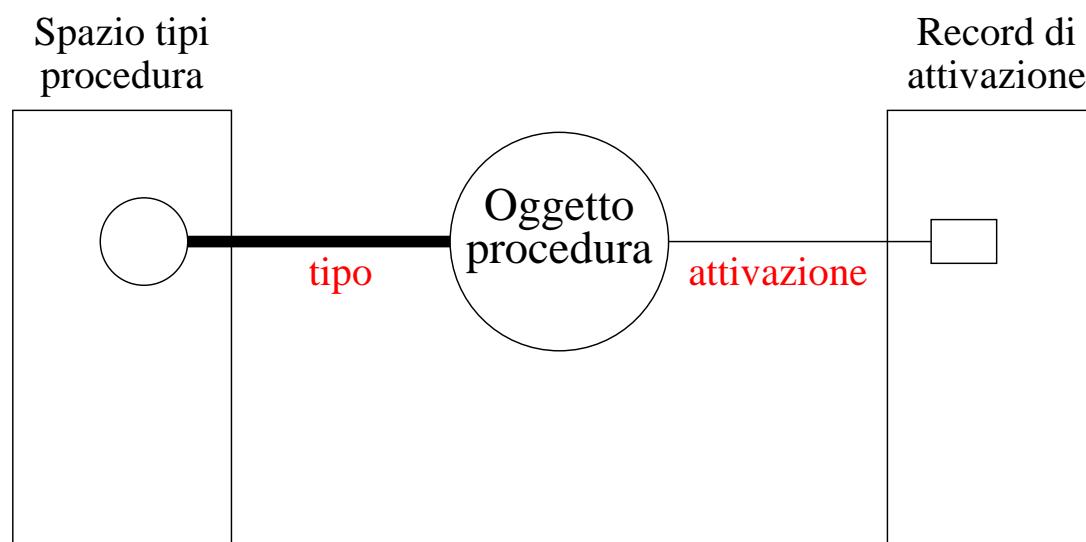
Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

- Causa la generazione di un oggetto analogo al Data Object.
- Il processo avviene durante l'esecuzione, nel momento in cui c'è l'invocazione della procedura.
- Ogni invocazione diversa della stessa procedura causa la generazione di un nuovo "oggetto procedura" con lo stesso legame di tipo, ma con diverso record di attivazione.





## Ambiente di esecuzione

Analogamente a quanto avveniva per un blocco (di cui la procedura è astrazione), il **record di attivazione** rappresenta l'intero ambiente di esecuzione di una procedura.

[Procedure come astrazioni](#)

---

[Procedure Astrazione procedurale](#)

[Dichiarazione](#)

[Invocazione di ...](#)

**Ambiente di ...**

[Esempio](#)

[Record di attivazione](#)

[Propagazione dei data object](#)

---

[Bibliografia](#)

---



## Ambiente di esecuzione

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

Analogamente a quanto avveniva per un blocco (di cui la procedura è astrazione), 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'interno della procedura);



## Ambiente di esecuzione

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

Analogamente a quanto avveniva per un blocco (di cui la procedura è astrazione), 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'interno della procedura);
2. ambiente non locale (tutti i data object la cui definizione è propagata da altre procedure);



## Ambiente di esecuzione

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

Analogamente a quanto avveniva per un blocco (di cui la procedura è astrazione), 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'interno della procedura);
2. ambiente non locale (tutti i data object la cui definizione è propagata da altre procedure);
3. ambiente dei parametri (contiene informazioni sui dati che sono passati [d]alla procedura).



## Ambiente di esecuzione

Procedure come astrazioni

Procedure Astrazione procedurale  
Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

Analogamente a quanto avveniva per un blocco (di cui la procedura è astrazione), 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'interno della procedura);
2. ambiente non locale (tutti i data object la cui definizione è propagata da altre procedure);
3. ambiente dei parametri (contiene informazioni sui dati che sono passati [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.



# Esempio

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei data object

Bibliografia

## Stack esecuzione

p: prima di chiamare s

p

```
program p;
var i;

procedure q;
begin
 ...
end;

procedure r;
begin
 i:= i-1;
 if i>0 then
 r
 else
 q
 end;

procedure s;
begin
 r;
 q
end;

begin
 i:= 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

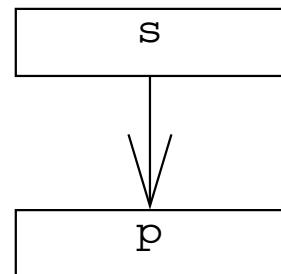
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

s: prima di chiamare r



```
program p;
var i;

procedure q;
begin
 ...
end;

procedure r;
begin
 i:= i-1;
 if i>0 then
 r
 else
 q
end;

procedure s;
begin
 r;
 q
end;

begin
 i:= 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

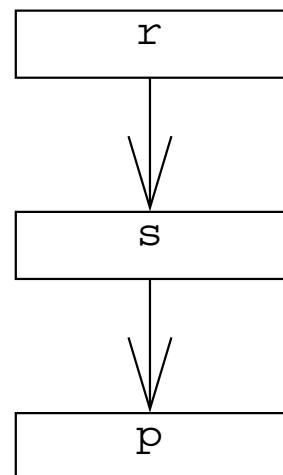
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

r: prima di chiamare r



program p;  
var i;  
  
procedure q;  
begin  
...  
end;  
  
procedure r;  
begin  
i:= i-1;  
if i>0 then  
r  
else  
q  
end;  
  
procedure s;  
begin  
r;  
q  
end;  
  
begin  
i:= 2;  
s  
end.



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

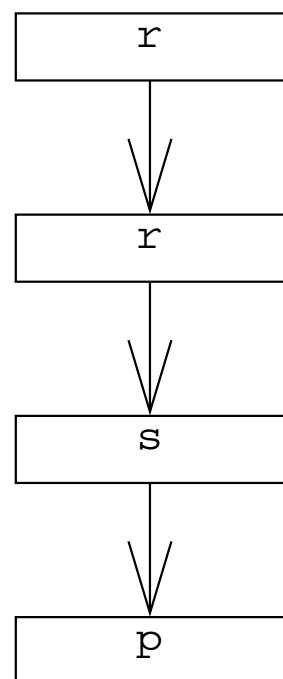
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

r: prima di chiamare q



program p;  
var i;  
  
procedure q;  
begin  
...  
end;  
  
procedure r;  
begin  
i := i-1;  
if i>0 then  
r  
else  
q  
end;  
  
procedure s;  
begin  
r;  
q  
end;  
  
begin  
i := 2;  
s  
end.

○

●



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

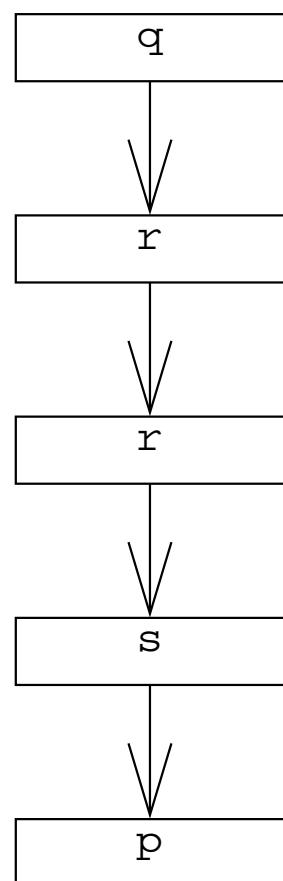
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

q: prima di terminare



```
program p;
var i;

procedure q;
begin
 ...
end;

procedure r;
begin
 i := i-1;
 if i>0 then
 r
 else
 q
 end;

procedure s;
begin
 r;
 q
end;

begin
 i := 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

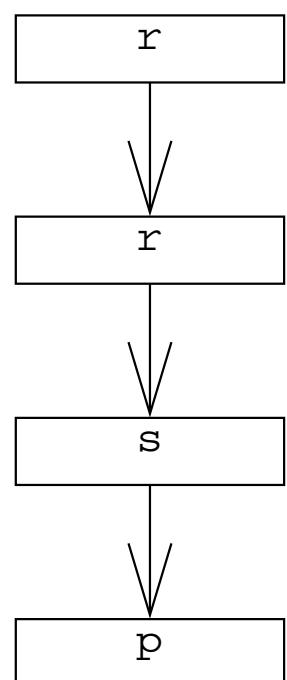
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

r: prima di terminare



program p;  
var i;  
  
procedure q;  
begin  
...  
end;  
  
procedure r;  
begin  
i := i-1;  
if i>0 then  
r  
else  
q  
end;  
  
procedure s;  
begin  
r;  
q  
end;  
  
begin  
i := 2;  
s  
end.



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

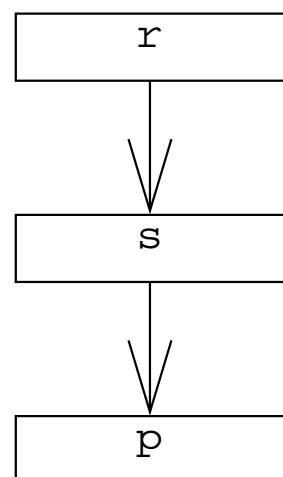
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

r: prima di terminare



```
program p;
var i;

procedure q;
begin
 ...
end;

procedure r;
begin
 i:= i-1;
 if i>0 then
 r
 else
 q
 end;

procedure s;
begin
 r;
 q
end;

begin
 i:= 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

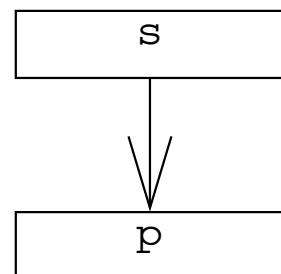
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

s: prima di q



```
program p;
var i;

procedure q;
begin
 ...
end;

procedure r;
begin
 i:= i-1;
 if i>0 then
 r
 else
 q
 end;

procedure s;
begin
 r;
 q
end;

begin
 i:= 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

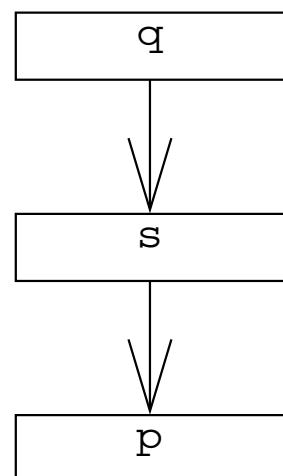
Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

q: prima di terminare



```
program p;
var i;

procedure q;
begin
 ...
end;

procedure r;
begin
 i:= i-1;
 if i>0 then
 r
 else
 q
 end;

procedure s;
begin
 r;
 q
end;

begin
 i:= 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

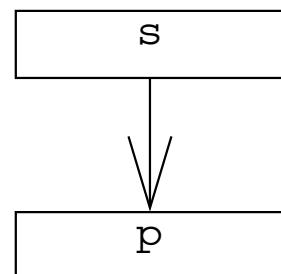
Record di attivazione

Propagazione dei data object

Bibliografia

## Stack esecuzione

s: prima di terminare



program p;  
var i;  
  
procedure q;  
begin  
...  
end;  
  
procedure r;  
begin  
i:= i-1;  
if i>0 then  
r  
else  
q  
end;  
  
procedure s;  
begin  
r;  
q  
end;  
  
begin  
i:= 2;  
s  
end.



# Esempio

Procedure come astrazioni

Procedure  
Astrazione  
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Esempio

Record di attivazione

Propagazione dei  
data object

Bibliografia

## Stack esecuzione

p: prima di terminare

p

```
program p;
var i;

procedure q;
begin
 ...
end;

procedure r;
begin
 i:= i-1;
 if i>0 then
 r
 else
 q
end;

procedure s;
begin
 r;
 q
end;

begin
 i:= 2;
 s
end.
```



Procedure come  
astrazioni

**Record di attivazione**

Ambiente locale

Esempio

Propagazione dei  
data object

Bibliografia

## Record di attivazione



# Ambiente locale

Procedure come astrazioni

Record di attivazione

Ambiente locale

Esempio

Propagazione dei  
data object

Bibliografia

Include:

1. Tutte le variabili dichiarate localmente.



# Ambiente locale

Procedure come astrazioni

Record di attivazione

Ambiente locale

Esempio

Propagazione dei  
data object

Bibliografia

Include:

1. Tutte le variabili dichiarate localmente.
2. Puntatore alla prossima istruzione [IP] (permette di riprendere l'esecuzione quando il controllo viene restituito alla procedura chiamante).



# Ambiente locale

Procedure come astrazioni

Record di attivazione

Ambiente locale

Esempio

Propagazione dei  
data object

Bibliografia

Include:

1. Tutte le variabili dichiarate localmente.
2. Puntatore alla prossima istruzione [IP] (permette di riprendere l'esecuzione quando il controllo viene restituito alla procedura chiamante).
3. Memoria temporanea necessaria alla valutazione delle espressioni contenute nella procedura (altamente dipendente dalla realizzazione).



# Esempio

Procedure come astrazioni

Record di attivazione

Ambiente locale

Esempio

Propagazione dei data object

Bibliografia

## Run time stack

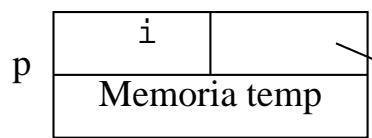
```
program p;
var i;

procedure q;
var vq;
begin
 ...
end;

procedure r;
var vr;
begin
 i:= i-1;
 vr:= i;
 if vr > 0 then
 r
 else
 q
 end;

procedure s;
var vs;
begin
 r;
 s
end;

begin
 i:= 2;
 s
end.
```





# Esempio

Procedure come astrazioni

Record di attivazione

Ambiente locale

Esempio

Propagazione dei data object

Bibliografia

## Run time stack

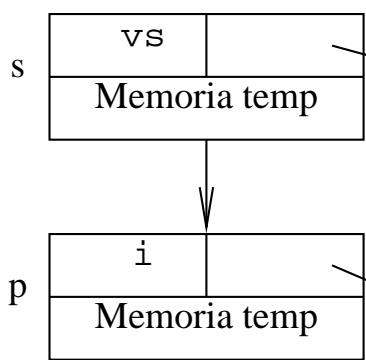
```
program p;
var i;

procedure q;
var vq;
begin
 . . .
end;

procedure r;
var vr;
begin
 i := i-1;
 vr := i;
 if vr > 0 then
 r
 else
 q
end;
```

```
procedure s;
var vs;
begin
 r;
 s
end;

begin
 i := 2;
 s
end.
```





# Esempio

Procedure come astrazioni

Record di attivazione

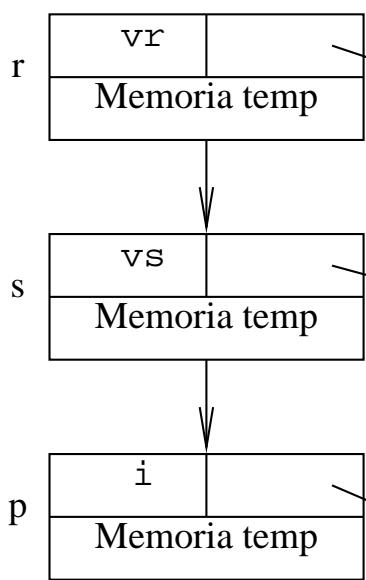
Ambiente locale

Esempio

Propagazione dei data object

Bibliografia

## Run time stack



```
program p;
var i;

procedure q;
var vq;
begin
 . . .
end;

procedure r;
var vr;
begin
 i := i-1;
 vr := i;
 if vr > 0 then
 r
 else
 q
 end;

procedure s;
var vs;
begin
 r;
 s
end;

begin
 i := 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Record di attivazione

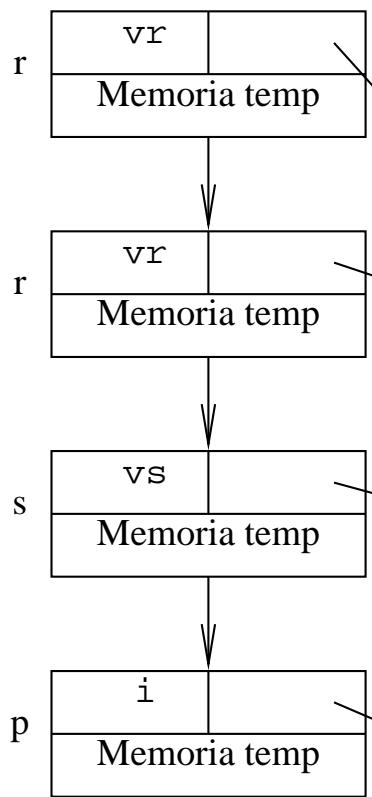
Ambiente locale

Esempio

Propagazione dei data object

Bibliografia

## Run time stack



```
program p;
 var i;

 procedure q;
 var vq;
 begin
 ...
 end;

 procedure r;
 var vr;
 begin
 i := i-1;
 vr := i;
 if vr > 0 then
 r
 else
 q
 end;

 procedure s;
 var vs;
 begin
 r;
 s
 end;

begin
 i := 2;
 s
end.
```



# Esempio

Procedure come astrazioni

Record di attivazione

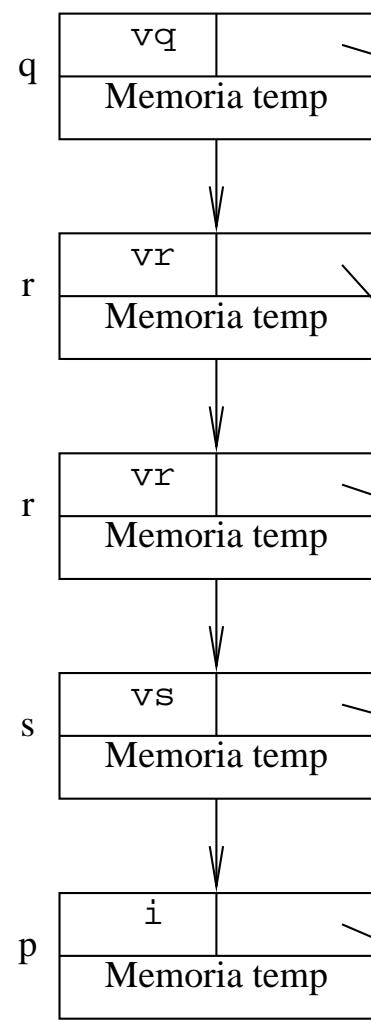
Ambiente locale

Esempio

Propagazione dei data object

Bibliografia

## Run time stack



```
program p;
var i;

procedure q;
var vq;
begin
 ...
end;

procedure r;
var vr;
begin
 i := i-1;
 vr := i;
 if vr > 0 then
 r
 else
 q
end;

procedure s;
var vs;
begin
 r;
 s
end;

begin
 i := 2;
 s
end.
```



Procedure come astrazioni

Record di attivazione

Propagazione dei data object

Realizzazione

Ambito statico

Ambito statico (2)

Ambito dinamico

Osservazioni

Bibliografia

## Propagazione dei data object



## Realizzazione

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



## Realizzazione

- Viene realizzata 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.



## Realizzazione

- Viene realizzata 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 realizzazione della propagazione.

- Viene realizzata 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 realizzazione della propagazione.
  1. Propagazione in ambito statico. In questo caso l'ambiente non locale di una procedura è propagato dal programma che la contiene sintatticamente: propagazione di posizione.

- Viene realizzata 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 realizzazione della propagazione.
  1. Propagazione in ambito statico. In questo caso l'ambiente non locale di una procedura è propagato dal programma che la contiene sintatticamente: propagazione di posizione.
  2. Propagazione in ambito dinamico. In questo caso l'ambiente non locale di una procedura è propagato dal programma chiamante.

- Viene realizzata 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 realizzazione della propagazione.
  1. Propagazione in ambito statico. In questo caso l'ambiente non locale di una procedura è propagato dal programma che la contiene sintatticamente: propagazione di posizione.
  2. Propagazione in ambito dinamico. In questo caso l'ambiente non locale di una procedura è propagato dal programma chiamante.
  3. Nessuna propagazione. L'uso di ambienti non locali è scoraggiato perché produce **effetti collaterali** non facilmente prevedibili.



# Ambito statico

Procedure come astrazioni

Record di attivazione

Propagazione dei data object

Realizzazione

Ambito statico

Ambito statico (2)

Ambito dinamico

Osservazioni

Bibliografia

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

 procedure q;
 var a, c; integer;
 procedure r;
 var a: integer;
 begin {r}
 ...
 end; {r}
 begin {q}
 ...
 end; {q}

 procedure s;
 var b: integer;
 begin {s}
 ...
 end; {s}

begin {p}
 ...
end. {p}
```

{variabili: a da r; b da p; c da q;  
procedure: q da p; r da q}

{variabili: a da q; b da p; c da q;  
procedure: q ed s da p; r da q}

{variabili: a da p; b da s; c da p;  
procedure: q ed s da p}

{variabili: a, b, c da p;  
procedure: q, s da p}



## Ambito statico (2)

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

[Procedure come astrazioni](#)

---

[Record di attivazione](#)

---

[Propagazione dei data object](#)

---

[Realizzazione](#)

[Ambito statico](#)

**Ambito statico (2)**

[Ambito dinamico](#)

[Osservazioni](#)

---

[Bibliografia](#)

---



## Ambito statico (2)

Procedure come astrazioni

Record di attivazione

Propagazione dei data object

Realizzazione

Ambito statico

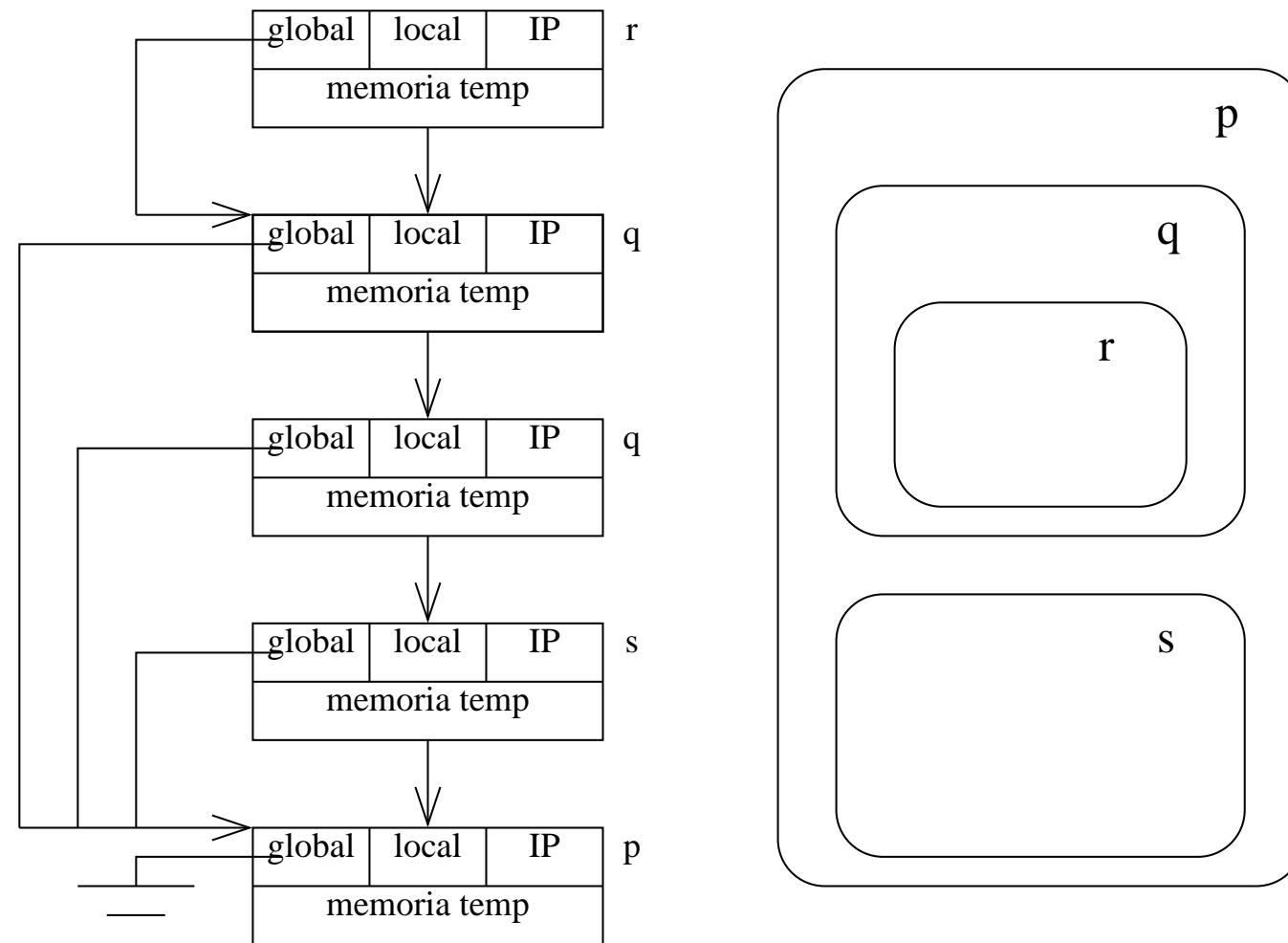
**Ambito statico (2)**

Ambito dinamico

Osservazioni

Bibliografia

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





## Ambito dinamico

Procedure come astrazioni

Record di attivazione

Propagazione dei data object

Realizzazione

Ambito statico

Ambito statico (2)

Ambito dinamico

Osservazioni

Bibliografia

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



# Ambito dinamico

Procedure come astrazioni

Record di attivazione

Propagazione dei data object

Realizzazione

Ambito statico

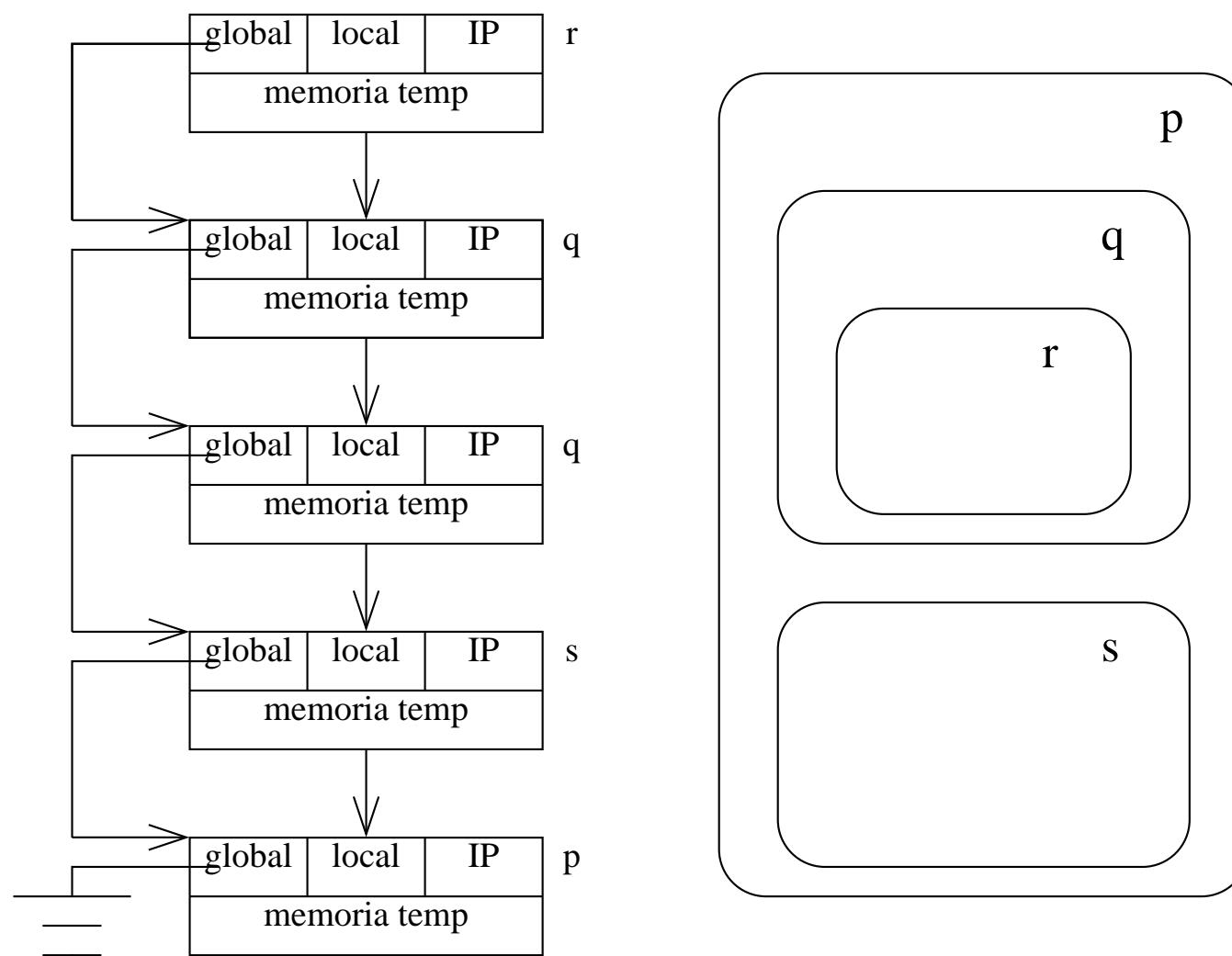
Ambito statico (2)

Ambito dinamico

Osservazioni

Bibliografia

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





## Osservazioni

Nella propagazione in ambito dinamico:

- il puntatore all'ambiente non locale non è più necessario;



## Osservazioni

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.



## Osservazioni

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.

```
program p;
var a: integer;
procedure q;
begin {vars: a da p o da r; procs: q da p}
 ...
end;
procedure r;
var
 a: integer;
begin {vars: a da r; procs: q, r da p}
 ...
end;
begin {vars: a da p; procs: q, r, da p}
 ...
end.
```



## Bibliografia

Procedure come astrazioni

Record di attivazione

Propagazione dei data object

Bibliografia

Bibliografia

- H. L. Dershem. M. J. Jipping. *Programming languages: structures and models*. Second edition. Cap. 5, par. 5.1 - 5.3.

Type equivalence  
+  
evoluzione dei sistemi di tipi  
+  
classificazione dei linguaggi di  
programmazione

# Ritorno al passato

- All'inizio esistevano solo i tipi elementari
  - E nessuna gerarchia di classi...
- Poi sono stati introdotti i tipi user-defined
  - Ancora niente classi
  - Inizialmente semplici ridenominazioni di tipi elementari oppure nomi di record
- Un assegnamento  $x = y$  (o  $x := y$ ) o un passaggio di parametri quando era valido?
- Lo stabilisce la nozione di *type equivalence* adottata dal linguaggio

# Forme di equivalenza

- **Name equivalence**

- I tipi di x e y devono avere lo stesso nome
- cioè essere lo stesso tipo

- **Structural equivalence**

- I tipi di x e y devono avere la stessa rappresentazione interna
- Apparentemente più snello e flessibile, in realtà aumenta la possibilità di errori

```
Euro x;
Dollar y;
z = x+y ; // che senso ha?
```

# Esempi

- Pascal: name equivalence
- C (e C++): entrambe!
  - ◆ Quasi sempre structural tranne che per le struct

```
typedef int money;
typedef int apples;

typedef struct{ int a; } S1;
typedef struct{ int a; } S2;

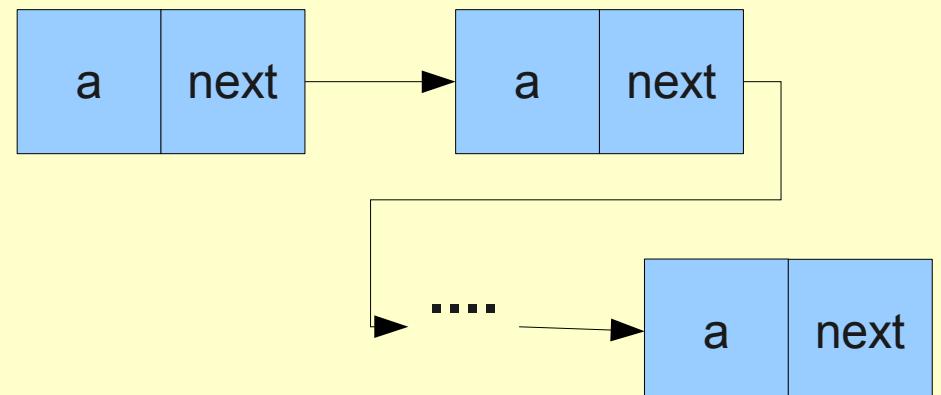
int main(){
 money x=0;
 apples y=0;
 int z = x+y; // non fa una piega
 S1 a;
 S2 b;
 a = b; // questo invece non lo compila
```

# Ecco perchè

- Verificare se due struct sono strutturalmente equivalenti richiede di verificare una proprietà chiamata *bisimulazione*
- Caso semplice: sia S1 che S2 generano tutte le catene come quella a destra

```
struct S1 { int a; struct S1* next; };

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



- Complicazioni in presenza di record varianti

# Compatibilità di tipi

- Con l'avvento dei linguaggi a oggetti l'equivalenza viene rimpiazzata da *compatibilità*
  - Qualunque sottotipo di T è compatibile con T
- Nei linguaggi O.O. più 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 sottotipo di un'altra deve essere *esplicitamente* dichiarata tale (keyword *extends*)
    - La struttura non conta neanche in questo caso

# Evoluzione dei sistemi di tipi

- Tipi di dato elementari
- Tipi user-defined (ad es. Pascal, C)
  - Solo strutture dati
- Interfacce e tipi di dato astratti (ad es. Modula, Ada)
  - Encapsulation: strutture dati accessibili solo attraverso specifiche procedure
  - Modificatori di accesso
  - Disaccoppiamento interfaccia/implementazione
- Gerarchie di tipi
  - Ed ecco finalmente i linguaggi O.O.

# Caratteristiche utili alla classificazione dei linguaggi

- Paradigma di riferimento
  - imperativo, O.O, funzionale, logico
- Scoping (statico/dinamico)
- Gestione della memoria
  - Allocazione dinamica, garbage collection/esplicita, ...
- Sistema di tipi
  - strong/weak, encapsulation, equivalence/compatibility, polimorfismo (di 4 tipi), *type inference*...
- Supporto alle eccezioni
  - *Eventualmente integrato con type checking... vedi ML*
- Supporto al parallelismo
  - Memoria condivisa (*synchronized*), scambio di messaggi (nel senso *RMI*), gestione del nondeterminismo, fairness

# Caratteristiche utili alla classificazione dei linguaggi

- Naturalmente determinano come si usa al meglio un dato linguaggio
- E quali sono gli errori da evitare
  - Ad es. sapendo cosa il compilatore può o non può fare per voi

# Linguaggi di Programmazione I – Lezione 9

Prof. Marcello Sette  
mailto://marcello.sette@gmail.com  
http://sette.dnsalias.org

6 maggio 2008

|                                 |           |
|---------------------------------|-----------|
| <b>Variabili</b>                | <b>3</b>  |
| Ambiti . . . . .                | 4         |
| Esempio . . . . .               | 5         |
| Inizializzazioni . . . . .      | 6         |
| <b>Espressioni</b>              | <b>7</b>  |
| Operatori . . . . .             | 8         |
| - logici . . . . .              | 9         |
| - di bit . . . . .              | 10        |
| - right shift . . . . .         | 11        |
| - left shift . . . . .          | 12        |
| Concatenazione . . . . .        | 13        |
| <b>Casting di primitivi</b>     | <b>14</b> |
| Conversioni implicite . . . . . | 15        |
| Conversioni esplicite . . . . . | 16        |
| Promozione aritm . . . . .      | 17        |
| <b>Enunciati branch</b>         | <b>18</b> |
| if, else . . . . .              | 19        |
| switch . . . . .                | 20        |
| <b>Enunciati loop</b>           | <b>21</b> |
| for . . . . .                   | 22        |
| while . . . . .                 | 23        |
| do/while . . . . .              | 24        |
| Controlli speciali . . . . .    | 25        |
| <b>Esercizi</b>                 | <b>26</b> |
| Esercizi . . . . .              | 26        |
| <b>Questionario</b>             | <b>27</b> |
| D 1 . . . . .                   | 28        |
| D 2 . . . . .                   | 29        |
| D 3 . . . . .                   | 30        |
| D 4 . . . . .                   | 31        |

## Panoramica della lezione

Variabili

Espressioni

Casting di primitivi

Enunciati branch

Enunciati loop

Esercizi

Questionario

LP1 – Lezione 9

2 / 31

## Variabili

3 / 31

### Ambiti

- Variabili definite all'interno di un metodo sono dette **locali**; esse sono create quando il metodo è invocato e sono 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 create quando viene costruito un oggetto. Ve ne sono di due tipi:
  - ◆ **Variabili di classe:** sono dichiarate usando il modificatore static; sono create nel momento in cui una classe è caricata in memoria (esistono a prescindere dalle istanze); continuano ad esistere finché la classe rimane in memoria; una variabile valida per tutte le istanze (discuteremo meglio in seguito).
  - ◆ **Variabili di istanza:** quelle dichiarate senza modificatore static; sono create durante la costruzione di una istanza; continuano ad esistere finché l'oggetto esiste; una variabile diversa per ciascuna istanza.

LP1 – Lezione 9

4 / 31

## Esempio

```
public class EsempioAmbito {
 private int i=1;

 public void primoMetodo() {
 int i=4, j=5;
 this.i = i + j;
 secondoMetodo(7);
 }

 public void secondoMetodo(int i) {
 int j=8;
 this.i = i + j;
 }
}

public class TestEsempioAmbito {
 public static void main (String[] args) {
 EsempioAmbito ambito = new EsempioAmbito();

 ambito.primoMetodo();
 }
}
```

## Inizializzazioni

- Nessuna variabile può essere usata prima di essere inizializzata!
- Le variabili non locali sono inizializzate dalla JVM, nel momento in cui 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        | '\u0000' |
| boolean     | false    |
| riferimenti | null     |

- Le variabili locali (quelle dei metodi) DEVONO essere inizializzate manualmente prima dell'uso. Esempio:

```
public void faQualcosa() {
 int x=(int)(Math.random()*9);
 int y;
 int z;
 if (x > 4) {
 y = 9;
 }
 z = y + x;
 // Errore: possibile uso
 // prima della
 // inizializzazione
}
```

## Operatori

Simili a quelli di C o C++. In ordine decrescente di precedenza:

|                                   |    |    |     |     |            |
|-----------------------------------|----|----|-----|-----|------------|
| Separatori                        | .  | [] | ()  | ;   | ,          |
| <b>Associat.</b> <b>Operatori</b> |    |    |     |     |            |
| R → L                             | ++ | -- | +   | -   | ~          |
| L → R                             | *  | /  | %   |     |            |
| L → R                             | +  | -  |     |     |            |
| L → R                             | << | >> | >>> |     |            |
| L → R                             | <  | >  | <=  | >=  | instanceof |
| L → R                             | == | != |     |     |            |
| L → R                             | &  |    |     |     |            |
| L → R                             | ^  |    |     |     |            |
| L → R                             |    |    |     |     |            |
| L → R                             | && |    |     |     |            |
| L → R                             |    |    |     |     |            |
| R → L                             | ?: |    |     |     |            |
| R → L                             | =  | *= | /=  | %=  | + = - =    |
|                                   | &= | ^= | =   | <<= | >>= >>>=   |

### - logici

- Operatori booleani: ! (NOT), & (AND), | (OR), ^ (XOR).
- Operatori booleani con corto circuito: || (OR), && (AND).
- Sono usati come segue:

```
int i = 1;
if (i) // genera un errore
if (i != 0) // corretto

MiaData d;
if ((d != null) && (d.giorno > 31)) {
 // manipola d
}
```

## - di bit

- Operatori di manipolazioni di bit su interi: `~` (Complemento ad 1), `&` (AND), `|` (OR), `^` (XOR).
- Esempi su byte

```
byte a = 45;
byte b = 79;
System.out.println(~b = " + (byte)(~b)); // -80
System.out.println("a&b = " + (byte)(a&b)); // 13
System.out.println("a^b = " + (byte)(a^b)); // 98
System.out.println("a|b = " + (byte)(a|b)); // 111
```

|   |                               |                                 |
|---|-------------------------------|---------------------------------|
|   |                               | 0   0   1   0   1   1   0   1   |
| ~ | 0   1   0   0   1   1   1   1 | & 0   1   0   0   1   1   1   1 |
|   | <hr/>                         | <hr/>                           |
|   | 1   0   1   1   0   0   0   0 | 0   0   0   0   1   1   0   1   |
|   |                               |                                 |
|   |                               | 0   0   1   0   1   1   0   1   |
| ^ | 0   1   0   0   1   1   1   1 | 0   1   0   0   1   1   1   1   |
|   | <hr/>                         | <hr/>                           |
|   | 0   1   1   0   0   0   1   0 | 0   1   1   0   1   1   1   1   |

## - right shift

- Nello shift a destra aritmetico o con segno (`>>`) il bit di segno viene conservato:
  - ◆ `128 >> 1` vale 64.
  - ◆ `128 >> 3` vale 16.
  - ◆ `-128 >> 3` vale -16.
- Nello shift a destra logico o senza segno (`>>>`) il bit di segno è azzerato nello spostamento:
  - ◆ `-1 >>> 30` vale 3.
- Se l'operando di sinistra è un `int`, prima di applicare lo shift viene ridotto l'operando di destra modulo 32; se l'operando di sinistra 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 `int` e `long`; se viene usato per `short` o `byte`, questi valori sono promossi, con estensione di segno, ad `int` prima di applicare lo shift.

## - left shift

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

- `128 << 1` vale 256.
- `16 << 3` vale 128.
- `-1 << 1` vale -2.
- `-1 << -31` vale -2.

## Concatenazione di stringhe

- L'operatore + esegue la concatenazione di due oggetti String, producendo un nuovo oggetto String.
- Esempio:

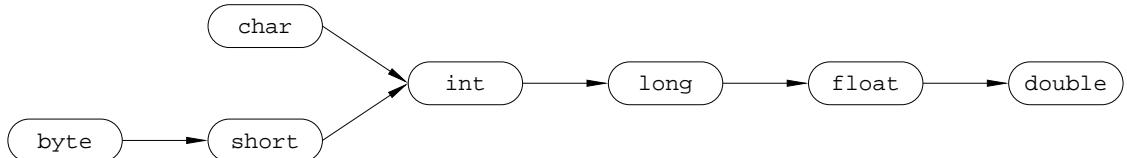
```
String titolo = "Dr.";
String nome = "Valentino" + " " + "Rossi";
String nomecompleto = titolo + " " + nome;
```

- Un argomento deve essere un oggetto String.
- Gli altri argomenti sono convertiti a String automaticamente, invocando il metodo `toString`.

## Casting di primitivi

### Conversioni implicite

- 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 boolean non può essere convertito MAI.
- Le conversioni legali (cioè quelle che avvengono automaticamente) sono quelle per cui esiste un percorso nel grafo:



### Conversioni esplicite

- **Regola:** 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.
- Per esempio, l'assegnazione da long a int richiede un cast esplicito:

```
long grande = 77L;
int piccolo = grande; // Sbagliato, richiede cast
int piccolo = (int) grande; // OK

int piccolo = 77L; // Sbagliato, richiede cast
int piccolo = (int) 77L; // OK, ma ...
int piccolo = 77; // pure OK
 // (default: 77 litterale int)
```

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

```
int i = 12;
byte b = 12; // OK
byte b = 128; // Illegale: e' superiore a 127
byte b = i; // Illegale: vale 12 ma non e' un litterale
float x = 3.14; // Illegale: eccezione non valida per f.p.
```

## 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 o 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 a double.
  - ◆ Se uno degli operandi è un float, allora l'altro operando è convertito a float.
  - ◆ Se uno degli operandi è un long, allora l'altro operando è convertito a long.
  - ◆ Altrimenti, entrambi gli operandi sono convertiti ad int.
- Che succede nella valutazione delle espressioni seguenti:

```
short s=9;
int i=10;
float f=11.1f;
double d=12.2;
if (-s * i >= f / d) {}
```

## Enunciati branch

if, else

```
if (<boolean expression>) {
 <statement>*
}

if (<boolean expression>) {
 <statement>*
} else if (<boolean expression>) {
 <statement>*
} else {
 <statement>*
}
```

Attenzione all'espressione condizionale: DEVE ESSERE UNA ESPRESSIONE BOOLEANA, NON UN INTERO COME IN C/C++.

## switch

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

dove:

- <expr> deve essere compatibile per assegnazione con int (byte, short, char sono promossi automaticamente);
- l'etichetta opzionale <default> è usata per specificare il segmento di codice da eseguire quando il valore di <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.

LP1 – Lezione 9

20 / 31

## Enunciati loop

21 / 31

### for

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

Esempio:

```
for (int i=0; i<10; i++) {
 System.out.println("Sono nel loop");
}
System.out.println("Fine");
```

LP1 – Lezione 9

22 / 31

### while

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

Esempio:

```
int i=0;
while (i<10) {
 System.out.println("Sono nel loop");
 i++;
}
System.out.println("Fine");
```

LP1 – Lezione 9

23 / 31

do/while

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

Esempio:

```
int i=0;
do {
 System.out.println("Sono nel loop");
 i++;
} while (i<10);
System.out.println("Fine");
```

LP1 – Lezione 9

24 / 31

## 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 qualunque enunciato verso il quale può essere trasferito il controllo.

LP1 – Lezione 9

25 / 31

## Esercizi

26 / 31

### Esercizi

1. Estensione dell'esercizio della lezione precedente (pacchetto banca).

LP1 – Lezione 9

26 / 31

## D 1

In questa successione di enunciati, quale linea non compila?

- A. byte b = 5;
- B. char c = '5';
- C. short s = 55;
- D. int i = 555;
- E. float f = 555.5f;
- F. b = s;
- G. i = c;
- H. if (f > b)
- I. f = i;

LP1 – Lezione 9

28 / 31

## D 2

Il codice seguente viene compilato correttamente?

```
byte b = 2;
byte b1 = 3;
b = b * b1;
```

- A. Si
- B. No

LP1 – Lezione 9

29 / 31

## D 3

Nel codice seguente, quali sono i possibili tipi per la variabile result?

```
byte b = 11;
short s = 13;
result = b * ++s;
```

- A. byte, short, int, long, float, double
- B. boolean, byte, short, char, int, long, float, double
- C. byte, short, char, int, long, float, double
- D. byte, short, char
- E. int, long, float, double

LP1 – Lezione 9

30 / 31

## D 4

```
1. class Cruncher {
2. void crunch(int i) {
3. System.out.println("int version");
4. }
5. void crunch(String s) {
6. System.out.println("String version");
7. }
8.
9. public static void main(String args[]) {
10. Cruncher crun = new Cruncher();
11. char ch = 'p';
12. crun.crunch(ch);
13. }
14. }
```

Quale delle seguenti affermazioni è vera?

- A. La linea 5 non compila, poiché i metodi void non possono essere sovrapposti.
- B. La linea 12 non compila, poiché nessuna versione di crunch() ha un argomento char.
- C. Il codice compila correttamente, ma viene lanciata una eccezione alla linea 12.
- D. Il codice compila correttamente e viene prodotto il seguente output: int version
- E. Il codice compila correttamente e viene prodotto il seguente output: String version