Corso di Linguaggi di Programmazione + Laboratorio
Docente: Marco de Gemmis

Capitolo 1 - Introduzione

Si ringraziano il Prof. Giovanni Semeraro e il Dott. Pasquale Lops per la concessione del materiale didattico

Apprendimento di un linguaggio di programmazione

- Perché un insegnamento “Linguaggi di Programmazione”? Non sarebbe stato sufficiente quello di “Programmazione”?

Apprendimento di un linguaggio di programmazione

- Una delle competenze fondamentali di un *buon* informatico è quella di apprendere un nuovo linguaggio di programmazione con naturalezza e velocità
- Questa competenza non la si acquisisce soltanto imparando *ex novo* molti linguaggi diversi
  - conoscere tanti linguaggi di programmazione oppure conoscere i fondamenti che sono alla base dei linguaggi di programmazione?

Apprendimento di un linguaggio di programmazione

- Lingue “naturali”
  - esistono analogie, somiglianze derivanti dalla genealogia
  - Es.: italiano – rumeno
    - cioccolata = ciocolata
    - insalata = salata
    - treno = tren
- Linguaggi di programmazione
  - è difficile conoscerne un gran numero in modo approfondito
  - è possibile conoscerne a fondo i meccanismi che ne ispirano il *progetto* e l’*implementazione*

Lo studio degli aspetti generali dei linguaggi di programmazione è un passaggio chiave della formazione universitaria e professionale di un informatico

Macchine Astratte e Implementazione dei linguaggi di programmazione

- Qual è il significato dell'espressione: "implementazione di un linguaggio di programmazione"?
- Si tratta di un concetto strettamente correlato a quello di:

MACCHINA ASTRATTA

Macchine Astratte e Implementazione dei linguaggi di programmazione

- Calcolatore = macchina fisica
  - consente di eseguire algoritmi opportunamente formalizzati perché siano “comprensibili” all’esecutore
  - la formalizzazione consiste nella codifica degli algoritmi in un certo linguaggio $\mathcal{L}$ definito da una specifica sintassi
  - la sintassi di $\mathcal{L}$ permette di utilizzare determinati costrutti per comporre programmi in $\mathcal{L}$
  - Un programma in $\mathcal{L}$ è una sequenza di istruzioni del linguaggio $\mathcal{L}$
- Macchina astratta = astrazione del concetto di calcolatore fisico

Definizioni di macchina astratta e linguaggio macchina

- Macchina astratta per $\mathcal{L}$
  - Un insieme di algoritmi e strutture dati che permettono di memorizzare ed eseguire programmi scritti in $\mathcal{L}$
  - Si denota con $\mathcal{M}_{\mathcal{L}}$
  - E' composta da:
    - una memoria per immagazzinare dati e programmi
    - un interprete che esegue le istruzioni contenute nei programmi
- Linguaggio Macchina
  - Data una macchina astratta $\mathcal{M}_{\mathcal{L}}$,il linguaggio $\mathcal{L}$ "compreso" dall'interprete di $\mathcal{M}_{\mathcal{L}}$ è detto linguaggio macchina di $\mathcal{M}_{\mathcal{L}}$

Le operazioni dell'Interprete

- Operazioni per l'elaborazione dei dati primitivi
  - numeri interi, reali
  - operazioni aritmetiche
- Operazioni e strutture dati per il controllo della sequenza di esecuzione
  - servono per gestire il flusso di controllo delle istruzioni
  - strutture dati per memorizzare l'indirizzo della prossima istruzione
  - operazioni per manipolare le strutture dati
    - es.: calcolo dell'indirizzo della prossima istruzione

Le operazioni dell'Interprete

- Operazioni e strutture dati per il controllo del trasferimento dei dati
  - gestiscono il trasferimento dei dati dalla memoria all'interprete
    - es.: recupero degli operandi
  - possono far uso di strutture dati ausiliarie
    - es.: pila
- Operazioni e strutture dati per la gestione della memoria
  - relative all'allocazione di memoria per dati e programmi

Ciclo di esecuzione del generico interprete

1. Acquisisci prossima istruzione da eseguire
2. Decodifica
   - determina l'operazione richiesta dall'istruzione e gli operandi
1. Preleva gli operandi ed esegui l'operazione richiesta
2. Memorizza l'eventuale risultato
3. Torna al punto 1 a meno che l'operazione non sia un HALT

Realizzazione di una macchina astratta

- Per essere effettivamente utilizzata, $\mathcal{M}_{\mathcal{L}}$ dovrà prima o poi utilizzare qualche dispositivo fisico
  - realizzazione “fisica” in hardware
    - algoritmi di $\mathcal{M}_{\mathcal{L}}$ realizzati direttamente mediante dispositivi fisici
- Possiamo pensare a realizzazioni che usino livelli intermedi tra $\mathcal{M}_{\mathcal{L}}$ ed il dispositivo fisico
  - **simulazione mediante software**
  - emulazione mediante firmware
    - microprogrammi in linguaggi di basso livello

Realizzazione mediante software

- Algoritmi e strutture dati di $\mathcal{M}_{\mathcal{L}}$ realizzati in un altro linguaggio $\mathcal{L}'$ già implementato
- $\mathcal{M}_{\mathcal{L}}$ è realizzata mediante programmi in $\mathcal{L}'$ che simulano le funzionalità di $\mathcal{M}_{\mathcal{L}}$
- $\mathcal{M}_{\mathcal{L}}$ è realizzata attraverso la macchina $\mathcal{M}'_{\mathcal{L}'}$
- $\mathcal{M}'_{\mathcal{L}'}$ è detta *macchina ospite* e si denota con $\mathcal{M}o_{\mathcal{L}o}$
- Intuitivamente l'implementazione di $\mathcal{L}$ sulla macchina ospite avviene mediante una qualche "*traduzione*" di $\mathcal{L}$ in $\mathcal{L}o$
- A seconda di come avvenga la traduzione di parla di:
  - **implementazione interpretativa**
  - **interpretazione compilativa**

Definizione di interprete

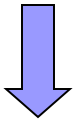
In generale un programma scritto in $\mathcal{L}$ si può vedere come una funzione parziale: $\mathcal{P}^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D}$ t.c. $\mathcal{P}^{\mathcal{L}}(Input) = Output$

Possiamo dare la seguente definizione di interprete di $\mathcal{L}$ in $\mathcal{L}o$:

$$I_{\mathcal{L}}^{\mathcal{L}o} : (Prog^{\mathcal{L}} \times D) \rightarrow D \qquad \text{tale che}$$

$$I_{\mathcal{L}}^{\mathcal{L}o}(\mathcal{P}^{\mathcal{L}}, Input) = \mathcal{P}^{\mathcal{L}}(Input)$$

Non vi è traduzione esplicita dei programmi scritti in $\mathcal{L}$, ma solo un procedimento di decodifica



L'interprete, per eseguire un'istruzione *i* di $\mathcal{L}$, le fa corrispondere un insieme di istruzioni di $\mathcal{L}o$. Tale decodifica non è una traduzione esplicita poiché il codice corrispondente a *i* è eseguito direttamente e non prodotto in uscita

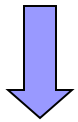
Definizione di compilatore

Un compilatore da $\mathcal{L}$ a $\mathcal{L}o$ è un programma che realizza la funzione:

$$C_{\mathcal{L},\mathcal{L}o} : Prog^{\mathcal{L}} \rightarrow Prog^{\mathcal{L}o}$$

$$C_{\mathcal{L},\mathcal{L}o}(\mathcal{P}^{\mathcal{L}}) = \mathcal{P}c^{\mathcal{L}o} \qquad \mathcal{P}^{\mathcal{L}}(\mathcal{D}) = \mathcal{P}c^{\mathcal{L}o}(\mathcal{D})$$

Traduzione esplicita dei programmi scritti in $\mathcal{L}$ in programmi scritti in $\mathcal{L}o$

$\mathcal{L}$ linguaggio sorgente  $\mathcal{L}o$ linguaggio oggetto

Per eseguire $\mathcal{P}^{\mathcal{L}}$ sull'input $\mathcal{D}$, bisogna eseguire $C_{\mathcal{L},\mathcal{L}o}$ con $\mathcal{P}^{\mathcal{L}}$ come input. Si avrà come risultato un programma compilato $\mathcal{P}c^{\mathcal{L}o}$ scritto in $\mathcal{L}o$, che sarà eseguito su $\mathcal{M}o_{\mathcal{L}o}$ con il dato di input $\mathcal{D}$

Interpretazione vs. Compilazione

- Interpretazione
  - scarsa efficienza
  - maggiore flessibilità
- Compilazione
  - maggiore efficienza: la decodifica di un'istruzione è fatta una sola volta indipendentemente da quante volte è eseguita
  - minore flessibilità: perdita di informazioni rispetto al programma sorgente (difficile tracciare errori e dunque maggiori difficoltà nel debugging)

In realtà i due approcci sono spesso combinati

Gerarchie di Macchine Astratte

- Possiamo pensare ad una gerarchia di macchine astratte $\mathcal{M}_{\mathcal{L}0}$, $\mathcal{M}_{\mathcal{L}1}$, … $\mathcal{M}_{\mathcal{L}n}$
- $\mathcal{M}_{\mathcal{L}i}$ è implementata sfruttando il linguaggio della macchina sottostante $\mathcal{M}_{\mathcal{L}i-1}$
- $\mathcal{M}_{\mathcal{L}i}$ fornisce a sua volta il proprio linguaggio alla macchina sovrastante $\mathcal{M}_{\mathcal{L}i+1}$
- Indipendenza tra livelli
  - modifiche *interne* alle funzionalità di un livello non hanno influenza sugli altri livelli

Una gerarchia di macchine astratte

Introduzione alla teoria dei linguaggi formali

- La teoria dei linguaggi formali rappresenta spesso uno scoglio per gli studenti di informatica a causa della sua forte dipendenza dalla notazione.
- L'argomento, d'altra parte, non può essere evitato perché ogni laureato in informatica deve possedere una buona comprensione dell'operazione di compilazione e questa, a sua volta, si fonda pesantemente sulla teoria dei linguaggi formali.

Introduzione alla teoria dei linguaggi formali

- Livelli di descrizione di un linguaggio
  - grammatica
    - quali frasi sono corrette?
  - semantica
    - cosa significa una frase corretta?
  - pragmatica
    - come usare una frase corretta e sensata?
  - implementazione (per i linguaggi di programmazione)
    - come eseguire una frase corretta in modo da rispettarne il significato?

Concetto intuitivo di grammatica

- Alfabeto del linguaggio
  - simboli con cui “costruire” le parole del linguaggio
  - es.: alfabeto latino su 22 o 26 lettere
- Lessico
  - parole del linguaggio
  - es.: informatica
- Sintassi
  - determina quali sequenze di parole costituiscono frasi legali (corrette)
- **Le grammatiche sono uno strumento utile per descrivere la sintassi di un linguaggio di programmazione**

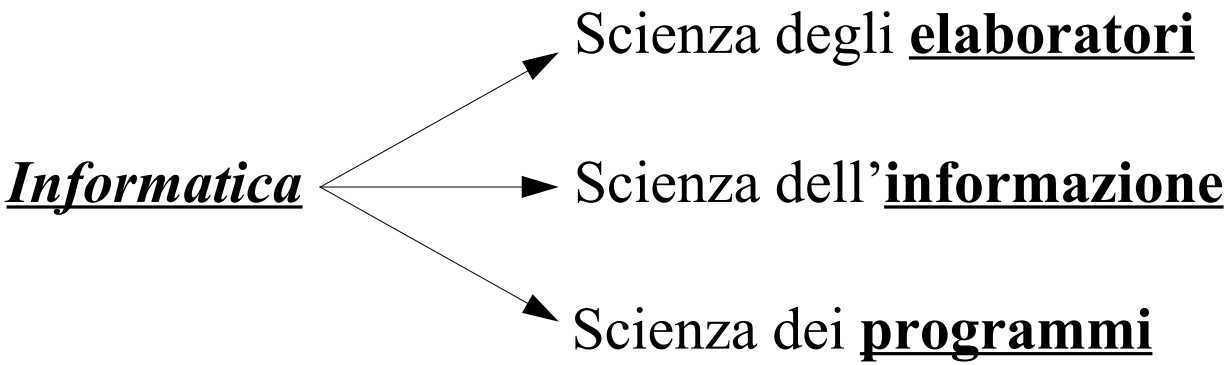
Introduzione alla teoria dei linguaggi formali

- Questa prima parte del programma ha l'obiettivo di fornire una base sufficiente alla comprensione delle tecniche di compilazione, oggetto della seconda parte del corso.
- E' per questa ragione che ci concentreremo quasi esclusivamente su due tipi di grammatiche:
  - regolari
  - libere da contesto

  poiché i linguaggi formali utilizzati in informatica sono per lo più di questi tipi.

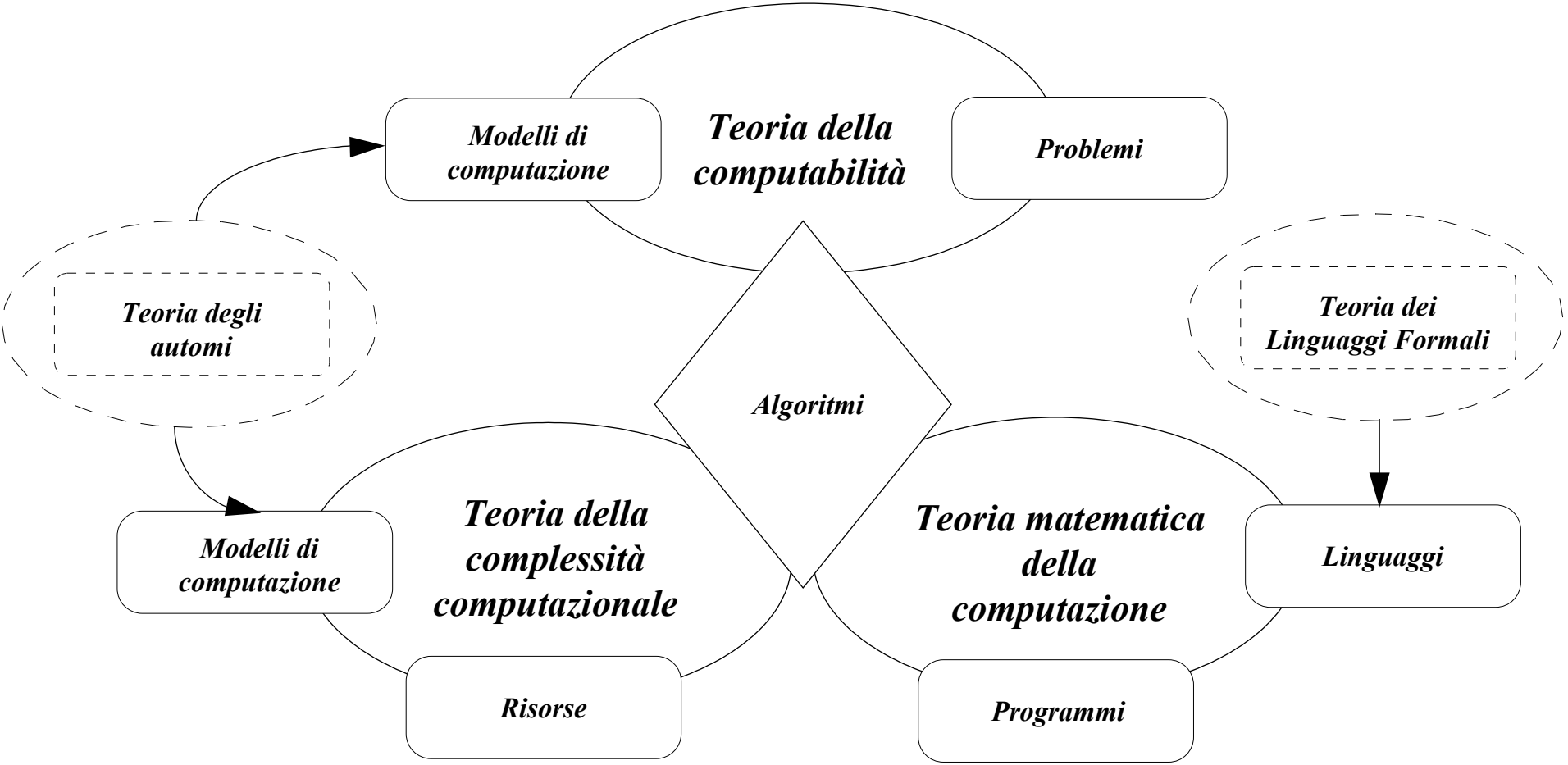
Introduzione

- L'informatica teorica è la **scienza degli algoritmi**, in tutti i loro aspetti, poiché è il concetto di algoritmo che è in qualche modo comune a tutti i settori dell'informatica.



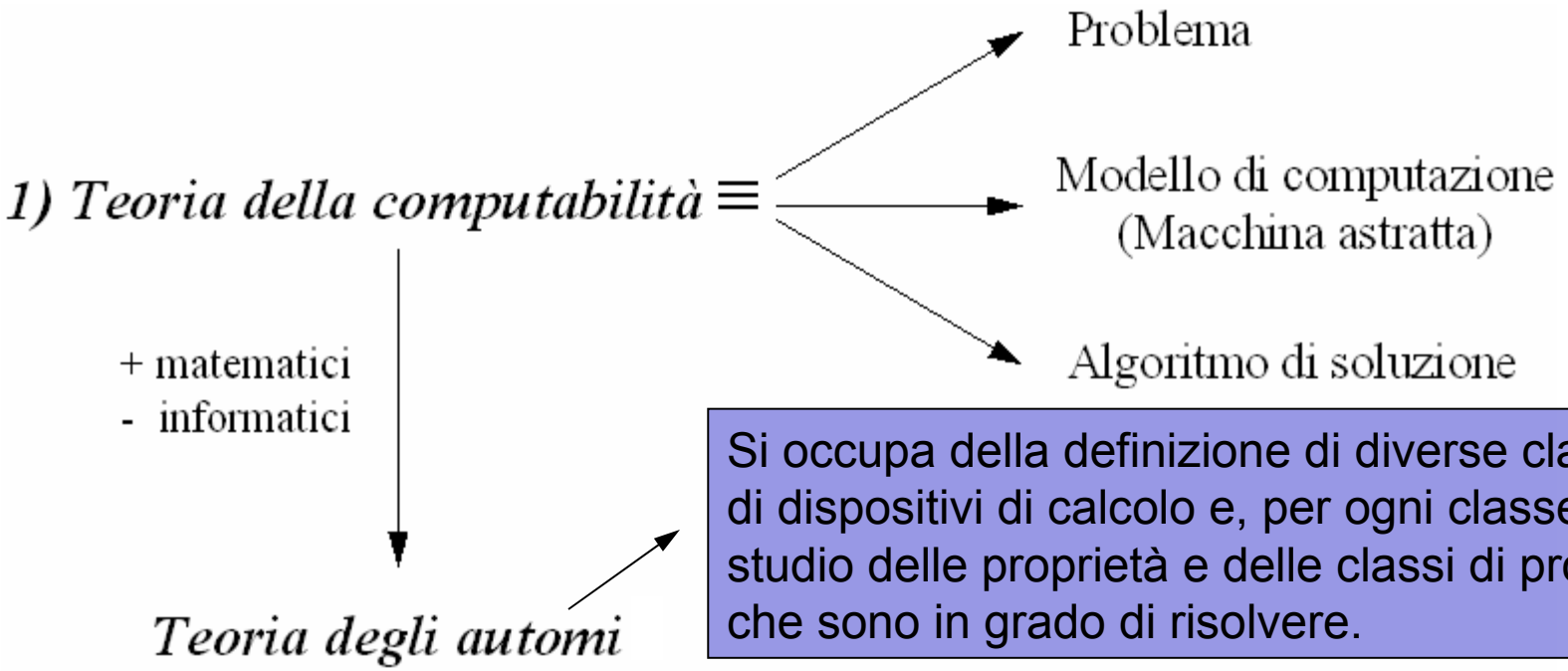
elaboratori	≡	macchine che eseguono gli algoritmi
informazione	≡	materia su cui lavorano gli algoritmi
programmi	≡	algoritmi descritti in un particolare linguaggio

Aree di ricerca dell'informatica teorica



Aree di ricerca dell'informatica teorica

- Lo schema riportato in figura fornisce una panoramica delle aree di ricerca che ricadono nell'ambito più generale dell'informatica teorica. Tali aree sono:



Aree di ricerca dell'informatica teorica

- Particolarmente feconda si è rivelata l'interazione tra la ***Teoria degli Automi*** e la ***Teoria dei Linguaggi Formali***.
- Quest'ultima, nata in ambito linguistico per caratterizzare i linguaggi naturali, è stata sviluppata ed utilizzata dagli informatici teorici, che avevano l'esigenza di descrivere gli algoritmi con linguaggi di programmazione comprensibili dalla macchina, ma non troppo difficili per l'uomo e soprattutto non ambigui.

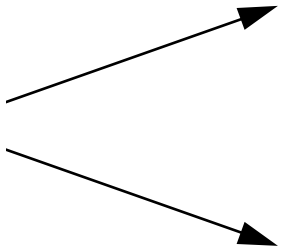
Aree di ricerca dell'informatica teorica

2) Teoria matematica della computazione ≡

- Rapporto tra gli algoritmi e i linguaggi di programmazione (in cui possono essere descritti)
- Proprietà dei programmi (correttezza, terminazione, ...)

Aree di ricerca dell'informatica teorica

3) Teoria della complessità computazionale ≡



Rapporto tra gli algoritmi e le risorse necessarie per eseguirli

Aspetti quantitativi del procedimento di soluzione di un problema

Studio dei linguaggi

- Quando iniziamo a studiare un linguaggio, formale o meno, tutti quanti godiamo di un grande vantaggio: siamo esperti in un linguaggio, quello con cui comunichiamo con gli altri.
- Oltre ad essere competenti in uno o più linguaggi naturali, lo studente in informatica ha familiarità con diversi linguaggi di programmazione, come il FORTRAN, il PASCAL, il C, il PROLOG, ...

Studio dei linguaggi

- Questi linguaggi sono usati per scrivere programmi e quindi per comunicare con il computer.
- Chiunque utilizzi un computer potrà notare che esso è particolarmente limitato nel comprendere il significato desiderato dei programmi.
- In linguaggio naturale possiamo di solito comunicare anche attraverso frasi mal strutturate e incomplete
- Quando dobbiamo comunicare con un computer la situazione cambia.

Sintassi e semantica

- I nostri programmi devono aderire rigorosamente a regole stringenti e anche differenze minori vengono rigettate come errate.
- Idealmente, ogni frase in un linguaggio dovrebbe essere corretta sia semanticamente (avere cioè il corretto significato) sia sintatticamente (avere la corretta struttura grammaticale).
- Nell'italiano parlato, le frasi sono spesso sintatticamente sbagliate, ciononostante convogliano la semantica desiderata.

Sintassi e semantica

- In programmazione, comunque, è essenziale che la sintassi sia corretta al fine di comunicare una qualsivoglia semantica.
- Quando studiamo la semantica di una frase, ne stiamo studiando il significato.

Sintassi e semantica

- Tutte le frasi che seguono hanno la stessa interpretazione semantica, cioè lo stesso significato, sebbene siano sintatticamente differenti:

The man hits the dog

The dog is hit by the man

L'homme frappe le chien

Sintassi e semantica

- Lo studio della sintassi è lo studio della grammatica, cioè della struttura delle frasi. La frase:

  The man hits the dog

  può essere analizzata sintatticamente, cioè risolta nelle parti grammaticali componenti, come segue:

  ed ogni frase in questa forma è sintatticamente valida in inglese.

Sintassi e semantica

- Possiamo descrivere un particolare insieme di tali frasi semplici in inglese usando le seguenti regole:

< frase semplice > :: =< parte nominale > < parte verbale >
< parte nominale >

< parte nominale > :: =< articolo > < nome >

< nome > :: = ***car*** | ***man*** | ***dog***

< articolo > :: = ***The*** | ***a***

< parte verbale > :: = ***hits*** | ***eats***

Sintassi e semantica

- Queste regole sono scritte in BNF, una notazione usata comunemente per descrivere la sintassi dei linguaggi di programmazione.
- BNF (Backus Naur Form) è un metalinguaggio.
- Nell'esempio, <*frase semplice*> è definita come una <*parte nominale*> seguita da una <*parte verbale*> seguita, a sua volta, da un'altra <*parte nominale*>.
- Ciascuna delle due occorrenze di <*parte nominale*> deve essere espansa in <*articolo*> seguito da un <*nome*>.

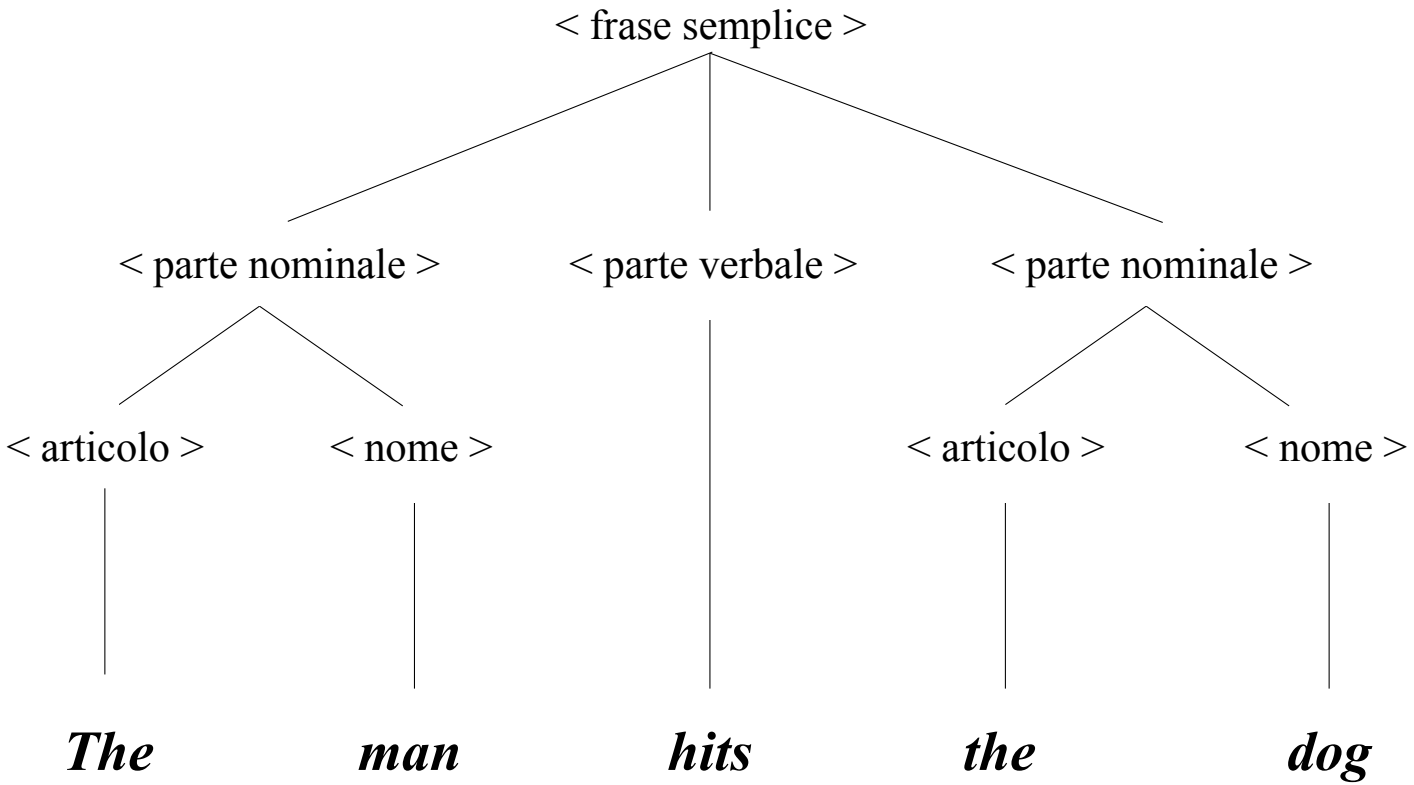
Sintassi e semantica

- Scegliendo di espandere la prima occorrenza di <*articolo*> in ***The***, la prima occorrenza di <*nome*> in ***man***, la <*parte verbale*> in ***hits***, il secondo <*articolo*> in ***The*** ed infine l'ultimo <*nome*> in ***dog***, si dimostra che la frase ***The man hits the dog*** è una delle nostre frasi semplici.

Sintassi e semantica

- Tutto ciò si può riassumere nel seguente

Albero di derivazione



Sintassi e semantica

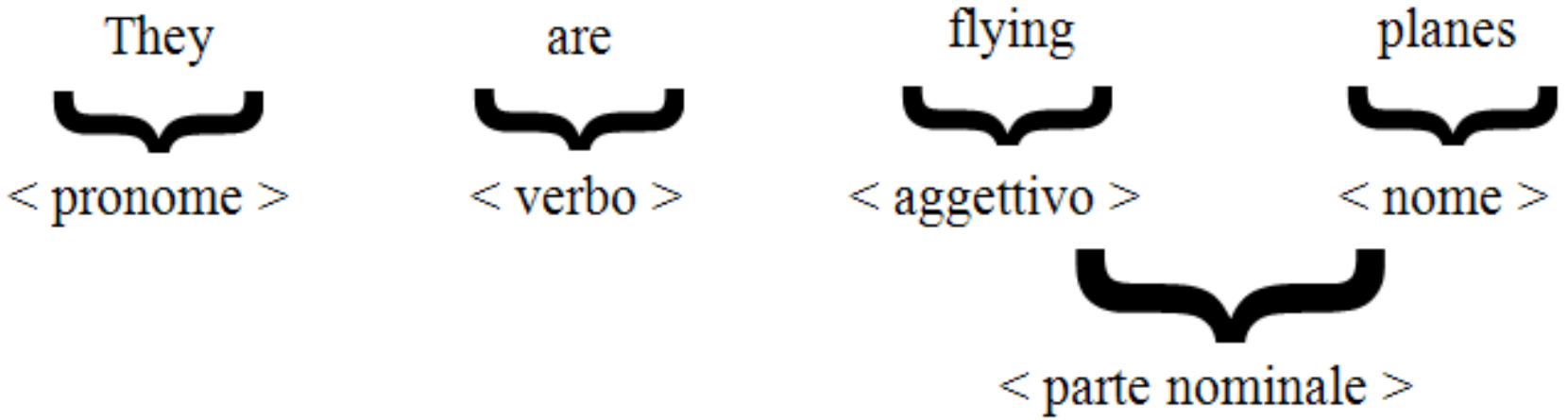
- La definizione data di <*frase semplice*> dà luogo a 72 diverse frasi derivabili.
  - Sebbene tutte siano sintatticamente corrette, in base alla nostra definizione, alcune non hanno un'interpretazione semantica sensata (non hanno senso compiuto).
  - Una di queste è la frase: ***The car eats the man***
  - Non necessariamente una frase corretta ha senso.

Sintassi e semantica

- Consideriamo la seguente frase (non derivabile da *<frase semplice>* utilizzando le regole date in precedenza):

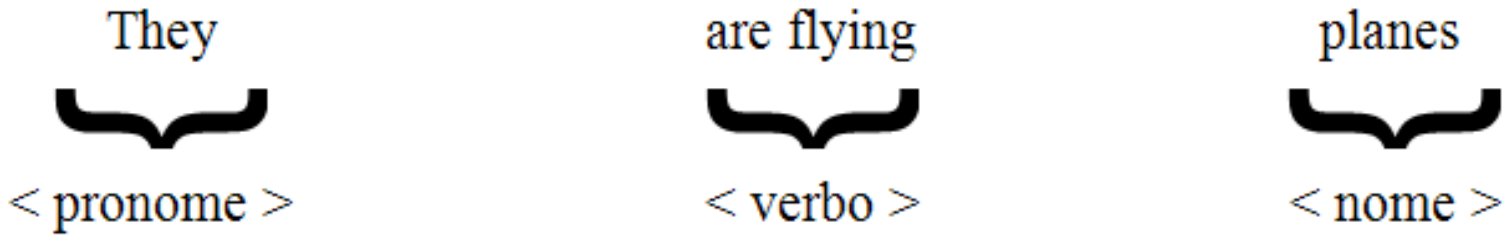
They are flying planes

- Un'analisi sintattica di questa frase è:



Sintassi e semantica

- Quest'analisi suggerisce che ***they*** si riferisce a ***planes*** (*aerei nel cielo che volano*). Un'analisi alternativa sarebbe:



e questa implicherebbe un'interpretazione semantica completamente differente, in cui ***they*** si riferisce a chiunque sia attualmente al controllo dei ***planes*** (*essi stanno pilotando gli aerei*).

Sintassi e semantica

- Nell'esempio, l'analisi sintattica è di aiuto all'interpretazione semantica.
- Per quanto sia scelto "ad hoc" e mostri un fenomeno poco comune nel linguaggio naturale, l'esempio è illustrativo di un concetto importante in computazione.
- Una fase cruciale nel processo di compilazione di un programma è l'analisi sintattica, come passo essenziale per la sua interpretazione semantica.

Teoria dei linguaggi formali

- Negli anni ‘50, N. Chomsky, linguista americano cerca di descrivere la sintassi del linguaggio naturale secondo semplici regole di riscrittura e trasformazione.
- Chomsky considera alcune restrizioni sulle regole sintattiche e classifica i linguaggi in base alle restrizioni imposte alle regole che generano tali linguaggi.
- Una classe importante di regole che generano linguaggi formali va sotto il nome di grammatiche libere da contesto (prive di contesto) o Context-free grammars (C.F.).

Teoria dei linguaggi formali

- L'importanza di tale classe (e dei linguaggi da essa generati) risiede nel fatto che lo sviluppo dei primi linguaggi di programmazione di alto livello (ALGOL 60), che segue di pochi anni il lavoro di Chomsky, dimostra che le grammatiche C.F. sono strumenti adeguati a descrivere la sintassi di base di molti linguaggi di programmazione.

Esempio di linguaggio C.F.

- Un linguaggio C.F. è il linguaggio delle parentesi ben formate che comprende tutte le stringhe di parentesi aperte e chiuse bilanciate correttamente.

  Esempio:

  ( ) è ben formata;

  (( ) ( )) è ben formata;

  (( ) ( ) non è ben formata.

- Questo linguaggio è fondamentale in informatica come notazione per contrassegnare il “raggio d’azione” nelle espressioni matematiche e nei linguaggi di programmazione

Linguaggio delle parentesi ben formate

- Definizione
  - i) La stringa $(\;)$ è ben formata;
  - ii) se la stringa di simboli *A* è ben formata, allora lo è anche (A);
  - iii) se le stringhe *A* e *B* sono ben formate, allora lo è anche la stringa *AB*.

Linguaggio delle parentesi ben formate

- In corrispondenza di questa definizione induttiva, possiamo considerare un sistema di riscrittura che genera esattamente l'insieme delle stringhe lecite di parentesi ben formate:

  (1) $S \rightarrow (\;)$

  (2) $S \rightarrow (S)$

  (3) $S \rightarrow SS$

- Le regole di riscrittura (1), (2) e (3) sono dette *produzioni* o *regole di produzione*.
  - Stabiliscono che "data una stringa, si può formare una nuova stringa sostituendo una *S* o con ( ) o con (*S*) o con *SS*".

Linguaggio delle parentesi ben formate

- Confrontando la definizione di parentesi ben formate e le tre produzioni, si osserva una corrispondenza diretta tra le parti i) e ii) della definizione e le produzioni (1) e (2). La produzione (3) è apparentemente diversa dalla parte (iii) della definizione. Abbiamo utilizzato simboli distinti A e B nella definizione induttiva per evidenziare il fatto che le due stringhe di parentesi ben formate che consideriamo non sono necessariamente uguali. Nella produzione corrispondente non c'è necessità di usare simboli distinti perché in $S \rightarrow SS$ le due *S* che compaiono a destra possono essere sostituite indipendentemente.
- Possiamo cioè applicare una produzione ad una delle due *S* (alla prima o alla seconda) indifferentemente. Il risultato non cambia.

Linguaggio delle parentesi ben formate

- Esempio: Generazione di $(( ))(( )( ))$
  - Primo modo:

$$S \underset{(3)}{\Rightarrow} SS \underset{(2)}{\Rightarrow} (S)S \underset{(1)}{\Rightarrow} (( ))S \underset{(2)}{\Rightarrow} (( ))(S) \underset{(3)}{\Rightarrow} (( ))(SS) \underset{(1)}{\Rightarrow} (( ))(( )S) \underset{(1)}{\Rightarrow} (( ))(( )( ))$$

  - La corrispondente sequenza di applicazione delle produzioni è:

$$(3) \rightarrow (2) \rightarrow (1) \rightarrow (2) \rightarrow (3) \rightarrow (1) \rightarrow (1)$$

Linguaggio delle parentesi ben formate

- Esempio: Generazione di $(( ))(( )( ))$
  - Secondo modo:

$$S \underset{(3)}{\Rightarrow} SS \underset{(2)}{\Rightarrow} S(S) \underset{(3)}{\Rightarrow} S(SS) \underset{(1)}{\Rightarrow} S(( )S) \underset{(1)}{\Rightarrow} S(( )( )) \underset{(2)}{\Rightarrow} (S)(( )( )) \underset{(1)}{\Rightarrow} (( ))(( )( ))$$

  - La corrispondente sequenza di applicazione delle produzioni è:

$$(3) \rightarrow (2) \rightarrow (3) \rightarrow (1) \rightarrow (1) \rightarrow (2) \rightarrow (1)$$

- Una sequenza di applicazioni di regole di produzione prende il nome di *derivazione*.

Notazione

- Nei due esempi precedenti abbiamo fatto uso della notazione $\alpha \underset{(n)}{\Rightarrow} \beta$
  - L'interpretazione è la seguente: "da α si produce direttamente β per effetto dell'applicazione della regola di riscrittura *n*". Ad esempio
    $$SS \underset{(2)}{\Rightarrow} (S)S$$
  - si legge: "*SS* produce direttamente $(S)S$ per effetto dell'applicazione della regola di riscrittura (2)".
- Le derivazioni precedenti (modi 1 e 2) vengono riassunte attraverso la notazione:

  $$S \overset{*}{\Rightarrow} (( ))(( )( ))$$

  che leggiamo come "S produce $(( ))(( )( ))$ "

Notazione

- Nelle regole di produzione si è fatto uso di due tipi di simboli: caratteri che possono apparire nelle derivazioni, ma non nelle stringhe finali, detti *simboli nonterminali* o *variabili* (nell'esempio, *S* è il solo nonterminale) e caratteri che possono apparire nelle stringhe finali, detti *simboli terminali* (nell'esempio, e sono i simboli terminali).
- In BNF si usa la seguente notazione:

S	<*S*>
$\rightarrow$	:: =

Riferimenti

- Maurizio Gabbrielli, Simone Martini. Linguaggi di Programmazione, Principi e paradigmi. McGraw-Hill
  - Capitoli 1-2
- Giovanni Semeraro. Appunti di Teoria dei Linguaggi Formali. Adriatica Editrice, Bari, 1996.
  - Capitolo 1