Informatica Grafica

Corso di Laurea in Ingegneria Edile – Architettura

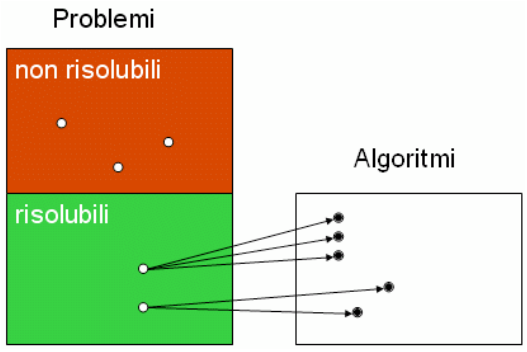
Fondamenti dell'Informatica e linguaggi di programmazione

Paolo Torroni

Dipartimento di Elettronica, Informatica e Sistemistica (DEIS)
Università degli Studi di Bologna

Anno Accademico 2011/2012

Fondamenti dell'Informatica e linguaggi di programmazione



- **Fondamenti dell'Informatica**
  - Procedimenti risolutivi (algoritmi)
  - Esistono problemi che nessun calcolatore è in grado di risolvere?
  - Esistono problemi risolubili in teoria ma non in pratica?
- **Linguaggi di programmazione**
  - Che linguaggi parlano i computer? Livelli di astrazione. Compilatori e interpreti.
  - Com'è fatto un linguaggio di programmazione?

Parte I

Concetti di base

Algoritmo per gli spaghetti all'arrabbiata

- **Ingredienti**: (per 6 persone)
  - spaghetti g 500
  - polpa di pomodoro fresco g 400
  - cipolla mondata g 30
  - olive nere
  - capperi
  - peperoncino piccante
  - basilico
  - olio d'oliva
  - sale

- Lessate gli spaghetti in abbondante acqua salata. Intanto, in una larga casseruola, fate riscaldare 3 cucchiaiate d'olio, insaporendolo con la cipolla tritata, un peperoncino intero, una decina di olive e una cucchiaiata di capperi. Unite quindi la polpa di pomodoro e fate cuocere il tutto per circa 5'. Scolate la pasta al dente e fatela saltare nel sugo, poi trasferitela in una terrina e servitela guarnita con basilico fresco.

Problemi, dati e risultati

- Tre elementi:
  - Scopo della ricetta: **problema** da risolvere
  - Ingredienti: **dati iniziali**
  - come trasformare gli ingredienti nel piatto finale: **istruzioni**
- Generalità:
  - generico cuoco, ingredienti
  - può essere ripetuta per ottenere lo stesso risultato
  - solo relazioni generali (es, 8 o 10 persone)

Proprietà degli algoritmi

- Generalità
- Comprensibilità da parte dell'esecutore
  - Uso di **operazioni elementari** alla portata dell'esecutore
- Finitezza
  - Numero limitato di istruzioni $\not\Rightarrow$ terminazione!
- Non ambiguità
  - Intanto fate riscaldare ...
  - Scolare la pasta al dente
    (Forse gli spaghetti all'arrabbiata non sono un buon esempio)
- Determinismo
  - Servire a piacere, con o senza parmigiano
- Efficienza (complessità)
- Semplicità

Algoritmo

Definizione (Algoritmo)

Successione non ambigua e ripetibile *di* **istruzioni eseguibili** *che permette di risolvere in modo* **generale** *un* **problema**.

- Soluzione costruita a partire da alcuni **dati iniziali**
- Istruzioni fanno riferimento a un'**insieme di operazioni elementari**, **eseguibili** da un certo **esecutore**.
- **Esecuzione** dell'algoritmo: seguire passo dopo passo le istruzioni

Un semplice problema: A+B

- Provare a scrivere **in linguaggio naturale** l'algoritmo per la somma di due numeri interi
  - Problema
  - Dati
  - Istruzioni
    1.
    2.
    3.

    ...
  - Quali sono le **operazioni elementari**?

Flusso del controllo

- Esecuzione “passo passo”
  1. Fate riscaldare 3 cucchiai d’olio . . .
  2. Unite la polpa di pomodoro
  3. Fate cuocere il tutto per 5 min
- Istruzione condizionale
  - **Se** la pasta è cotta **allora** scolare la pasta
    altrimenti aspettate 1 altro minuto
- Istruzione iterativa
  - **Continuate a** cuocere **finché** il sugo non ha raggiunto la consistenza desiderata

Algoritmi, programmi, calcolatori

- **Calcolatore**: esecutore di algoritmi
- **Linguaggio di programmazione**: linguaggio per esprimere algoritmi in modo comprensibile al calcolatore
- **Programma**: algoritmo scritto in un linguaggio di programmazione
- A ciascun calcolatore è associato un **linguaggio macchina**, cioè un linguaggio di programmazione che può comprendere
  - Calcolatore $C \Rightarrow$ Linguaggio macchina L_C
  - Programma P scritto in $L_C \Rightarrow P$ eseguibile da C
  - Dati P e opportuni dati di ingresso a $C \Rightarrow$ soluzione

Calcolatore Rudimentale (CR)

Composizione. CR consiste di due parti:

- **controllo**: esecuzione del programma;
- **memoria**: un numero **illimitato** di celle:
  - capaci di memorizzare un numero naturale;
  - indicate con un numero: C_1, C_2, C_3, ...

Operazioni elementari. 5 operazioni:

- **somma** 1 alla cella C_n;
- **memorizza** 0 nella cella C_n;
- **leggi** dall'esterno un numero, memorizzalo in C_n;
- **mostra** il contenuto della cella C_n;
- **stop**.

Flusso del controllo. Salto condizionato

- se C_n e C_m contengono lo stesso numero, vai all'istruzione (X).
  ...salto *incondizionato*?

Un esercizio di analisi

- Cosa fa il seguente programma?
  1. **Leggi** un numero e memorizzalo in C_1.
  2. **Memorizza** zero nella cella C_2.
  3. **Se** C_1 e C_2 contengono lo stesso numero, vai all'istruzione (5).
  4. **Somma** uno alla cella C_1.
  5. **Mostra** la cella C_1.
  6. **Stop**.

Un altro esercizio di analisi

- Cosa fa il seguente programma?
  1. **Leggi** un numero e memorizzalo in C_1.
  2. **Memorizza** zero nella cella C_2.
  3. **Se** C_1 e C_2 contengono lo stesso numero, vai all'istruzione (3).
  4. **Somma** uno alla cella C_1.
  5. **Mostra** la cella C_1.
  6. **Stop**.

Problema della terminazione

- La possibilità di avere programmi CR che non terminano è intrinsecamente legata alla definizione di CR.
- Costituisce una caratteristica fondamentale di ogni linguaggio di programmazione.

Un terzo programma da analizzare

- Cosa fa il seguente programma?
  1. **Leggi** un numero e memorizzalo in C_1.
  2. **Leggi** un numero e memorizzalo in C_2.
  3. **Memorizza** zero in C_3.
  4. **Se** C_2 e C_3 contengono lo stesso numero, vai all'istruzione (8).
  5. **Somma** uno alla cella C_1.
  6. **Somma** uno alla cella C_3.
  7. **Se** C_2 e C_2 contengono lo stesso numero, vai all'istruzione (4).
  8. **Mostra** la cella C_1.
  9. **Stop**.

Problemi CR-risolubili

- La somma è un problema CR-risolubile

Definizione (Problema CR-risolubile)

Un problema è CR-risolubile se esiste un programma CR che lo risolve.

- I problemi dell'aritmetica elementare ($A + B$, $A - B$, $A \times B$, $\frac{A}{B}$, A^B, $MCD(A, B)$, ...) sono tutti CR-risolubili.
- Esattamente quali problemi sono CR-risolubili?

Parte II

Problemi risolubili

Macchine e linguaggi più evoluti di CR

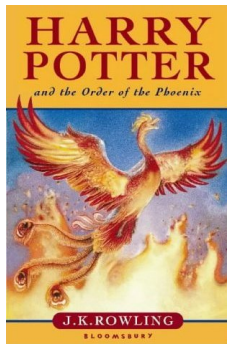
- Proviamo a esprimere algoritmi complessi nel linguaggio macchina di CR
  - migliaia di istruzioni
  - nessuna struttura che la colleghi alla struttura del problema originale
- Servono linguaggi per scrivere programmi concisi e strutturati (**linguaggi di alto livello**)
  - non ambigui
  - eseguibili
  - facilmente utilizzabili da un programmatore

Linguaggi di alto livello

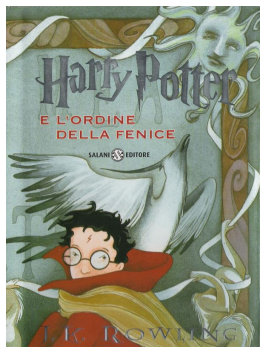
- Come rendere un linguaggio più semplice da utilizzare?
  - nomi simbolici per indicare le celle $\Rightarrow$ **variabili**
  - non solo numeri naturali $\Rightarrow$ **tipi di dato**
  - non solo somma $\Rightarrow$ **operatori**
  - non solo azzeramento $\Rightarrow$ **assegnamento**
  - non solo "Se $C_1 = C_2$ vai a (X)" $\Rightarrow$
    - **espressioni condizionali** (if, then, else) e
    - **iterazioni** (while, do) che usano
    - **espressioni logiche** $(=, <, \leq, \vee, \wedge, \ldots)$
  - possibilità di scomposizione in sottoproblemi $\Rightarrow$ **procedure**

Compilatori e interpreti

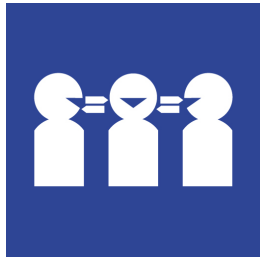
- Com'è possibile far eseguire a CR un programma scritto in un linguaggio di alto livello?
- **Traduzione**
  - **Compilazione**
    - In ingresso: il programma di alto livello P_L
    - In uscita: il programma per CR P_{CR}



$\Rightarrow$



  - **Interpretazione**
    - In ingresso: il programma di alto livello P_L e i dati per P_L
    - In uscita: il risultato dell'esecuzione di P_L



Problemi L-risolubili

Definizione (Problema L-risolubile)

Un problema è L-risolubile se esiste un programma scritto in L che lo risolve.

- Esistono problemi Java-risolubili, C-risolubili, ...
- Qual è la relazione tra la classe dei problemi L-risolubili e quella dei problemi CR-risolubili?
  - Classe più ampia/meno ampia/stessi elementi?

Tesi di Church-Turing

Teorema (Potenza espressiva dei linguaggi di programmazione)

Tutti i linguaggi di programmazione hanno la stessa potenza: risolvono gli stessi problemi, che possiamo identificare con quelli risolubili dal semplice calcolatore CR

- Tutti i linguaggi di programmazione progettati sinora hanno la stessa potenza
- Differenze tra linguaggi: ~~potenza espressiva~~, semplicità d'uso, facilità di apprendimento, eleganza, ...
  - Vero anche per i linguaggi che non sono stati ancora inventati?

Tesi (Tesi di Church-Turing)

Ogni problema che sia intuitivamente risolubile mediante un algoritmo è CR-risolubile

(principio empirico)

Parte III

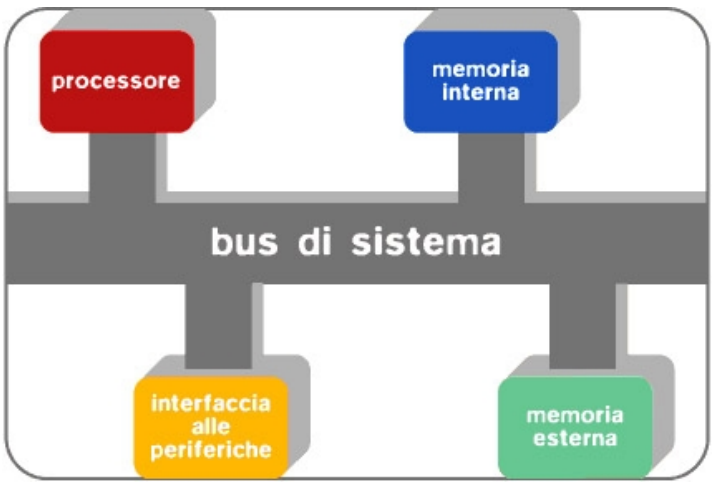
Problemi non risolubili

Macchina CR universale

- È possibile compiere operazioni aritmetiche tramite manipolazioni testuali
  - ASCII/Unicode: *carattere* $\leftrightarrow$ *numero*
- Interpretazione: il programma P_L è il dato di input
- Programma universale **UNIV**
  - Può simulare l'esecuzione del programma fornito come dato
  - Dati: il **testo** di un programma P e dei dati per P (D_P)
  - Esecuzione di **UNIV**$(P, D_P) \equiv$ esecuzione di $P(D_P)$
- **UNIV** è una **macchina universale**, cioè può simulare qualsiasi altro programma per CR.
- Il nostro **computer** è una macchina universale.
  - Può essere usato per eseguire programmi diversi
  - Interprete per L_{CR}
- Quindi basta fare riferimento all'architettura di **UNIV** per descrivere l'architettura di un generico computer

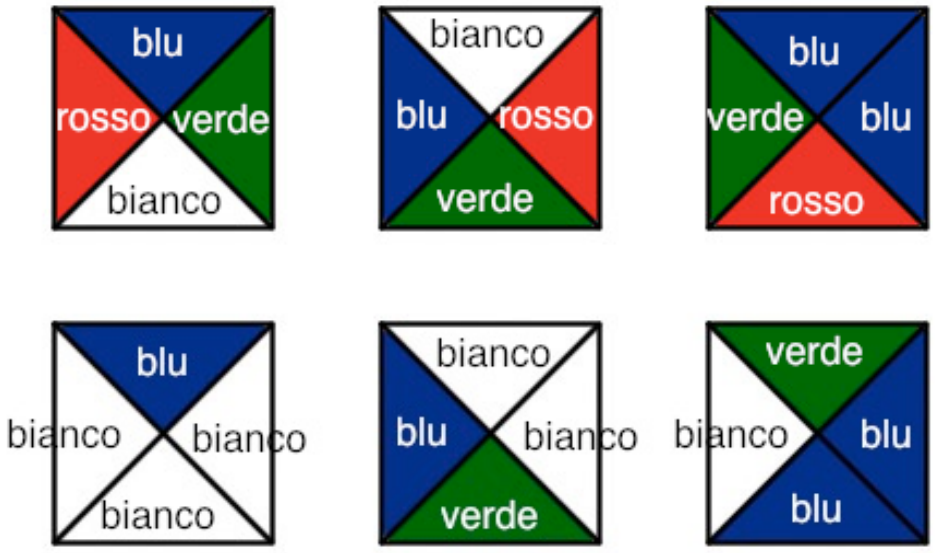
Architettura di Von Neumann

- Realizza l'universalità
  - Programma registrato in memoria
  - Dati registrati in memoria
  - Processore esegue un ciclo di interpretazione utilizzando programma e dati



Esistono problemi non CR-risolubili

- Tiling problem (Harel)
  - Dato: un insieme finito di tipi di piastrelle



  - Vincolo sul colore dei lati di piastrelle adiacenti
  - **Problema**: determinare se un tipo fornito in ingresso è in grado di piastrellare qualsiasi stanza, di qualsivoglia forma o dimensione.

Esistono problemi non CR-risolubili

Teorema (Irresolubilità assoluta del tiling problem)

È **impossibile** *scrivere un programma CR in grado di determinare se un tipo fornito in ingresso è in grado di piastrellare* **qualsiasi** *stanza, di qualsivoglia forma o dimensione.*

- Irresolubilità dipende dalla **struttura** del problema
- Motivazione intuitiva: non posso sapere se ho considerato **tutte** le possibili forme e dimensioni della stanza

Esistono problemi non CR-risolubili

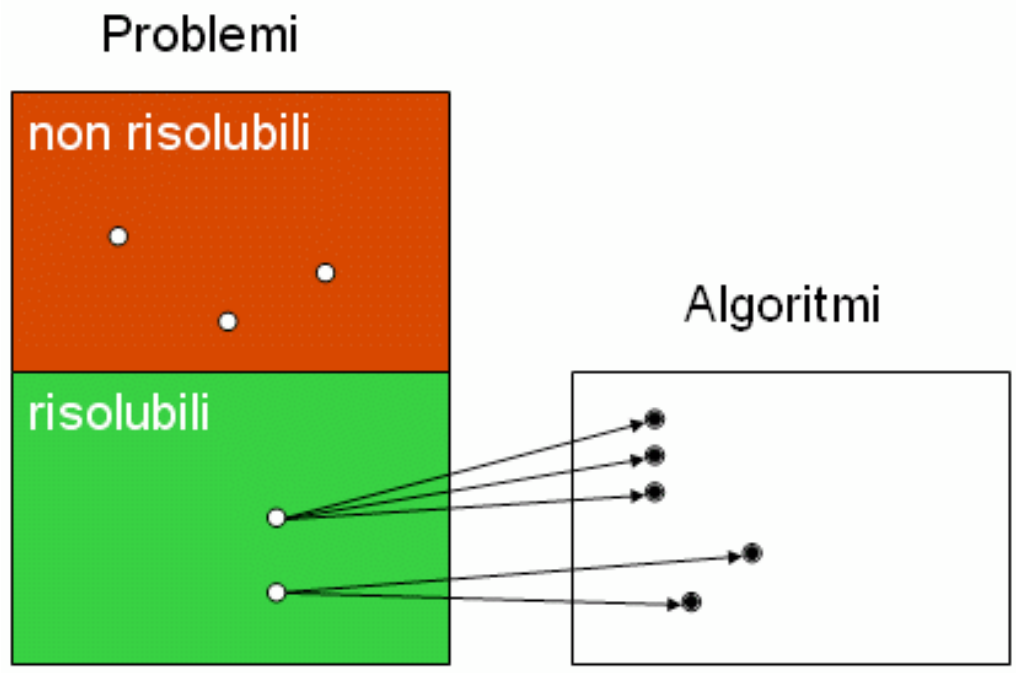
- Halting problem (Turing)
  - Esistono programmi che possono andare in ciclo.
  - Dato un programma, è possibile determinare che non andrà mai in ciclo per qualsiasi valore dei dati di ingresso?

Teorema (Irresolubilità assoluta dell'halting problem)

È **impossibile** *scrivere un programma CR che, preso come dato un altro (generico) programma P e un dato n per P, termina stampando SÌ se $P(n)$ termina; e termina stampando NO se $P(n)$ va in ciclo.*

Problemi non algoritmicamente risolubili

- Ogni problema algoritmicamente risolubile è CR-risolubile (Church-Turing)
- Esistenza di problemi non CR-risolubili: una limitazione assoluta, che **non dipende da CR** o dalla nostra capacità di inventare **linguaggi**
- Problemi non risolubili in senso assoluto: **indecidibili**



Parte IV

Problemi difficili da risolvere

Costo del calcolo

- Consideriamo solo i problemi CR-risolubili
- Quanto costa risolvere un problema di questa classe?
  - Costo = *tempo di calcolo*
  - Per indipendenza dal tipo di calcolare: **passi di calcolo**
  - Una qualsiasi operazione elementare costa una unità di tempo
  - Per semplicità: si guarda solo alla **dimensione** dei dati di ingresso, non al loro specifico valore
    - Somma: costo dipende dalla dimensione degli addendi
    - Minimo di un insieme: dipende dal numero di elementi
    - ...

Un esempio

- Consideriamo di nuovo il programma in L_{CR} per la somma:
  1. **Leggi** un numero e memorizzalo in C_1.
  2. **Leggi** un numero e memorizzalo in C_2.
  3. **Memorizza** zero in C_3.
  4. **Se** C_2 e C_3 contengono lo stesso numero, vai all'istruzione (8).
  5. **Somma** uno alla cella C_1.
  6. **Somma** uno alla cella C_3.
  7. **Se** C_2 e C_2 contengono lo stesso numero, vai all'istruzione (4).
  8. **Mostra** la cella C_1.
  9. **Stop**.
- Quanto costa sommare due numeri?
  - istruzioni (1)–(3): 3 passi iniziali
  - istruzioni (4)–(7): 4 passi *per ogni valore da 1 a* C_2
  - istruzioni (4): 1 passo *quando* $C_2 = C_3$
  - istruzione (8): 1 passo
- Costo: tempo di calcolo (**complessità**) $T(n) = 4 \times n + 5$
- $n =$ valore di C_2

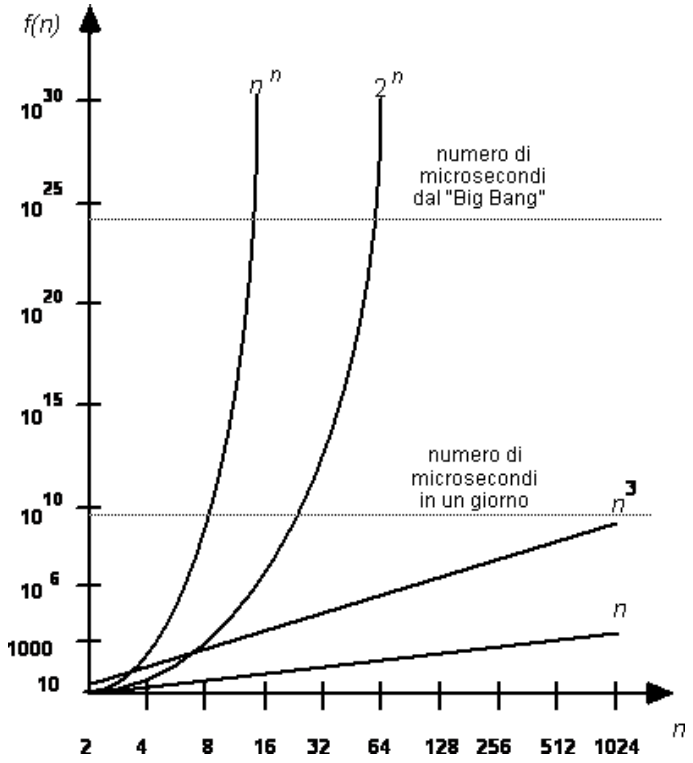
Tassi di crescita

- Interessa sapere l'andamento di $T(n)$ al crescere di n
- Esempio di prima: $T(n) \sim 4 \times n$ (complessità **lineare**)
- Anche: $T(n) = \mathcal{O}(n)$
- **Classi di complessità**

$\mathcal{O}(1)$	Complessità costante: $T(n) \sim K$
$\mathcal{O}(n)$	Complessità lineare: $T(n) \sim K \times n$
$\mathcal{O}(n^p)$	Complessità polinomiale: $T(n) \sim K \times n^p$
$\mathcal{O}(2^n)$	Complessità esponenziale: $T(n) \sim K \times 2^n$

Tempo esponenziale

- Cosa significa in pratica complessità esponenziale?
  - Ipotesi: un'operazione elementare $\Rightarrow 1\mu s = 10^{-6}s$
  - Complessità di P esponenziale: $T(n) = 2^n$
  - Dato di dimensione 50
  - Quanto tempo per ottenere la soluzione?



Trattabilità e intrattabilità

- Esiste una linea di confine tra algoritmi con complessità polinomiale ed esponenziale

Definizione (Problemi intrattabili)

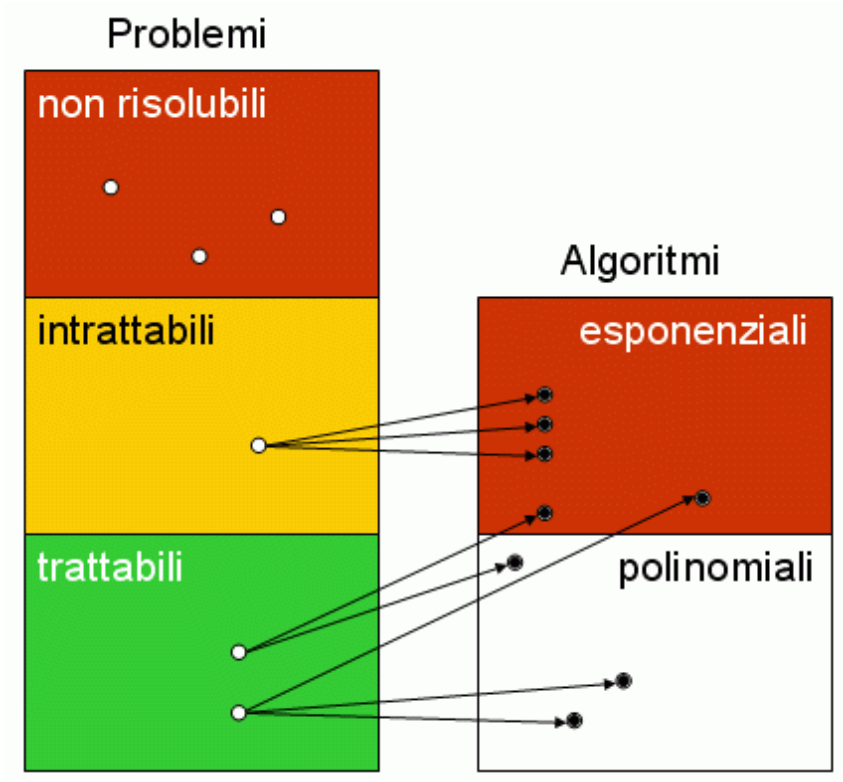
Un problema è intrattabile se **tutti gli algoritmi** *che lo risolvono hanno complessità per lo meno* **esponenziale**

Definizione (Problemi trattabili)

Un problema è trattabile se esiste per esso **almeno un algoritmo** *risolutivo con complessità* **polinomiale**

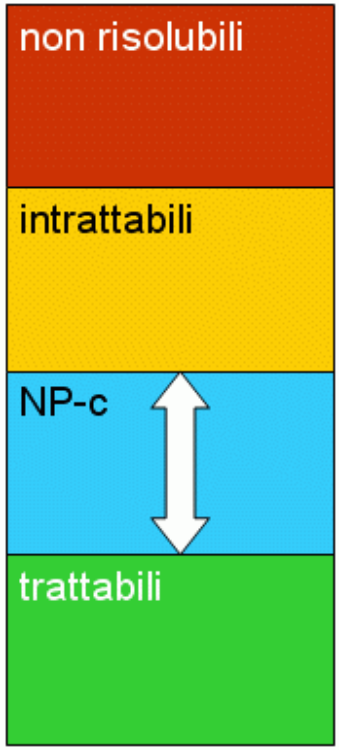
Trattabilità e intrattabilità

- Problemi intrattabili $\Rightarrow$ risolubili solo per dimensioni dei dati molto limitate.
- È sempre possibile inventare algoritmi inutilmente inefficienti



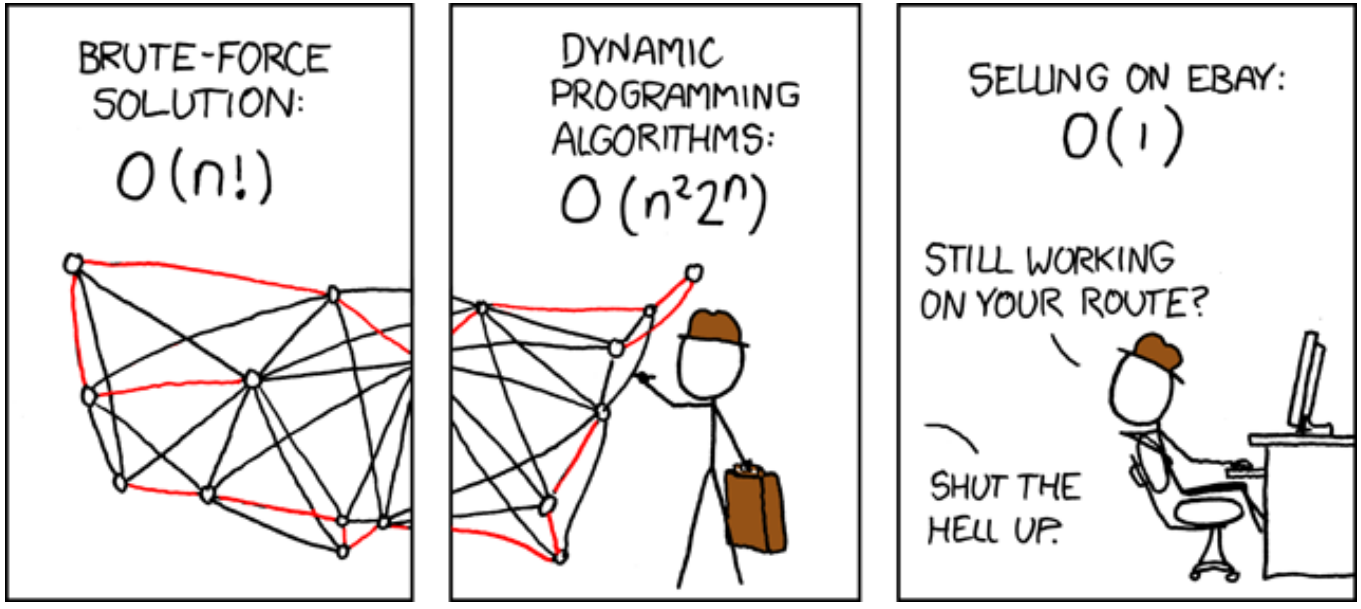
Alcuni esempi di problemi

- Problemi trattabili
  - Ordinamento ($nlogn$)
  - Cammino minimo (n^2)
  - Sottostringa
- Problemi non risolubili
  - Ricoprimento
  - Terminazione
- Problemi intrattabili
  - Ricoprimento finito (data una stanza specifica)



Problemi NP-completi

- Esistono problemi per cui
  - esiste una soluzione esponenziale
  - non è dimostrato che non esista una soluzione polinomiale
- Alcuni esempi:
  - Orario
  - Commesso viaggiatore



http://xkcd.com/399/

Parte V

Tipologie di linguaggi di programmazione

Dall'algoritmo al programma

- **Programma**:

  descrizione di un **algoritmo** *scritta in un* **linguaggio di programmazione**

- Caratteristiche di un linguaggio di programmazione:
  - facilità di utilizzo da parte dei programmatori
  - interpretabilità ed eseguibilità dal computer
  - assenza di ambiguità
  - chiarezza e concisione
    - poche strutture sintattiche
    - ciascuna con un significato ben preciso

Dall'algoritmo al programma

- **Programma**:

  rappresentazione eseguibile di un **algoritmo**
  progettato per risolvere un **problema**

- Elementi di un programma:
  - **Logica**: **cosa** vogliamo ottenere dal programma
  - **Controllo**: **come** vogliamo raggiungere tale scopo

$$\textbf{Algoritmo} = \textbf{logica} + \textbf{controllo}$$

- Ad esempio: definizione (logica) di $n!$ (per $n \geq 0$)

$$n! = \begin{cases} 1 & \textbf{se } n \leq 1 \\ n \times (n-1)! & \textbf{altrimenti} \end{cases}$$

- Dove sono logica e controllo?

Due categorie di linguaggi

- **Linguaggi imperativi**
  - definiscono insiemi di **operazioni elementari**
  - definiscono **come** devono essere eseguite (in che ordine e sotto quali condizioni)
  - attenzione sia alla logica sia al controllo

```
int i=1, fattoriale=1;
while( i<=N ) {
    fattoriale=fattoriale*i;
    i=i+1;
}
```

Due categorie di linguaggi

- **Linguaggi dichiarativi**
  - definiscono insiemi di **dichiarazioni** (relazioni tra fatti noti)
  - enfasi sul **cosa**, non sul **come**
  - attenzione soprattutto sulla logica

```
fattoriale(0,1).
fattoriale(1,1).

fattoriale(X,F) :-
    X1 is X-1,
    fattoriale(X1,F1),
    F is X*F1.
```

Linguaggi imperativi

- **Operazioni di base**
  - operazioni **aritmetiche** (somma, sottrazione, etc.)
  - operazioni di **ingresso e uscita** (es. da tastiera, su video)
- **Costrutti linguistici per il controllo di flusso**
  - **sequenza** (esegui A, **poi** esegui B)
  - **scelta** (**se** si verifica una certa condizione C, esegui A)
  - **iterazione**
    - esegui A **N volte**
    - continua a esegure A **finché** non si verifica C
- **Costrutti per lavorare su dati**
  - Possibilità di definire **variabili**
    - nome, tipo, utilizzo, operazioni possibili, ...
  - Operazione di **assegnamento**
  - Possibilità di **costruire** dati per aggregazione

Costruiamo un linguaggio (imperativo) per CR

Caratteristiche di un linguaggio: eseguibilità, compattezza, non ambiguità, facilità d'uso

⇒ **Eseguibilità**: partiamo dalle operazioni eseguibili da CR

Compattezza: nomi simbolici a operazioni e costrutti

Non ambiguità: esiste una unica interpretazione del codice?

Facilità d'uso: tipi di dato, nomi simbolici alle variabili, nuovi costrutti, possibilità di definirne in più.

Costruiamo un linguaggio (imperativo) per CR

Caratteristiche di un linguaggio: eseguibilità, compattezza, non ambiguità, facilità d'uso

$\Rightarrow$ **Eseguibilità**: partiamo dalle operazioni eseguibili da CR

- **somma** 1 alla cella C_n;
- **memorizza** 0 nella cella C_n;
- **leggi** dall'esterno un numero, memorizzalo in C_n;
- **mostra** il contenuto della cella C_n;
- **stop**;
- **se** C_n e C_m contengono lo stesso numero,vai all'istruzione (X).

Compattezza: nomi simbolici a operazioni e costrutti

Non ambiguità: esiste una unica interpretazione del codice?

Facilità d'uso: tipi di dato, nomi simbolici alle variabili, nuovi costrutti, possibilità di definirne in più.

Costruiamo un linguaggio (imperativo) per CR

Caratteristiche di un linguaggio: eseguibilità, compattezza, non ambiguità, facilità d'uso

Eseguibilità: partiamo dalle operazioni eseguibili da CR

- `INC` *`cell_num`*
- `RESET` *`cell_num`*
- `STORE` *`cell_num`*
- `WRITE` *`cell_num`*
- `HALT`
- `JMPEQ` *`cell_num_1 cell_num_2`* `(`*`instruction_num`*`)`

⇒ **Compattezza**: nomi simbolici a operazioni e costrutti

Non ambiguità: esiste una unica interpretazione del codice?

Facilità d'uso: tipi di dato, nomi simbolici alle variabili, nuovi costrutti, possibilità di definirne in più.

Costruiamo un linguaggio (imperativo) per CR

Caratteristiche di un linguaggio: eseguibilità, compattezza, non ambiguità, facilità d'uso

Eseguibilità: partiamo dalle operazioni eseguibili da CR

- `INC` *`cell_num`*
- `RESET` *`cell_num`*
- `STORE` *`cell_num`*
- `WRITE` *`cell_num`*
- `HALT`
- `JMPEQ` *`cell_num_1 cell_num_2`* (*`instruction_num`*)

Compattezza: nomi simbolici a operazioni e costrutti

⇒ **Non ambiguità**: esiste una unica interpretazione del codice?

Facilità d'uso: tipi di dato, nomi simbolici alle variabili, nuovi costrutti, possibilità di definirne in più.

Costruiamo un linguaggio (imperativo) per CR

Caratteristiche di un linguaggio: eseguibilità, compattezza, non ambiguità, facilità d'uso

Eseguibilità: partiamo dalle operazioni eseguibili da CR

- `int x, y;`
- `x=x+y;`
- `read( x, keyboard );`
- `write( screen, x );`
- ~~`HALT`~~
- `if( ( x>0 )AND( x+y < z ) ) { ... }`
- `while( x==y ) { ... }`
- `int fattoriale( x ) { ... }`

Compattezza: nomi simbolici a operazioni e costrutti

Non ambiguità: esiste una unica interpretazione del codice?

$\Rightarrow$ **Facilità d'uso**: tipi di dato, nomi simbolici alle variabili, nuovi costrutti, possibilità di definirne in più.

Eseguiamo un programma con CR

- **Linguaggio macchina** L_{CR} (direttamente eseguibile da CR)
- Linguaggio dei computer: `0` e `1`
- Necessario **codificare** le istruzioni in sequenze di `0` e `1`
- Codice "binario" (non "leggibile")
- CR è un calcolatore rudimentale $\Rightarrow$ possibili solo 6 istruzioni
  - Che formato per i dati?
  - Quanti bit servono?
  - Una possibile codifica?

Un possibile linguaggio macchina per CR

Tabella: Possibile codifica delle istruzioni di CR

`INC`	`001`	**somma** 1 alla cella C_n
`RESET`	`000`	**memorizza** 0 nella cella C_n
`STORE`	`010`	**leggi** dall'esterno un numero, memorizzalo in C_n
`WRITE`	`011`	**mostra** il contenuto della cella C_n
`HALT`	`111`	**stop**
`JMPEQ`	`100`	**se** $C_n = C_m$ vai all'istruzione (X)

Codifica degli operandi per CR

- Gli operandi si riferiscono a celle di memoria
- Che codifica in binario?
  - Quante celle di memoria?
  - Esempio: memoria grande 100 celle $\Rightarrow$ bastano 7 bit

Tabella: Codifica di istruzioni e operandi di CR

INC *cell_num*	001 xxxxxxx
RESET *cell_num*	000 xxxxxxx
STORE *cell_num*	010 xxxxxxx
WRITE *cell_num*	011 xxxxxxx
HALT	111
JMPEQ *c_1 c_2* (*i*)	100 xxxxxxx xxxxxxx xxxxxxx

Esempio di programma in linguaggio macchina

- Cosa fa il seguente programma?

```
1.  010 0001010
2.  010 0001011
3.  000 0001100
4.  100 0001011 0001100 0001000
5.  001 0001010
6.  001 0001100
7.  100 0001011 0001011 0000100
8.  011 0001010
9.  111
10.
11.
12.
```

Esempio di programma in linguaggio macchina

- Cosa fa il seguente programma?

```
1.  010 0001010
2.  010 0001011
3.  000 0001100
4.  100 0001011 0001100 0001000
5.  001 0001010
6.  001 0001100
7.  100 0001011 0001011 0000100
8.  011 0001010
9.  111
10.
11.
12.
```

- **Codice macchina** per CR (L_{CR})

Esempio di programma in linguaggio macchina

- Cosa fa il seguente programma?

```
1.  STORE 0001010
2.  STORE 0001011
3.  RESET 0001100
4.  JMPEQ 0001011 0001100 (0001000)
5.  INC 0001010
6.  INC 0001100
7.  JMPEQ 0001011 0001011 (0000100)
8.  WRITE 0001010
9.  HALT
10.
11.
12.
```

Esempio di programma in linguaggio macchina

- Cosa fa il seguente programma?

```
1.  STORE 10
2.  STORE 11
3.  RESET 12
4.  JMPEQ 11 12 (8)
5.  INC 10
6.  INC 12
7.  JMPEQ 11 11 (4)
8.  WRITE 10
9.  HALT
10.
11.
12.
```

Esempio di programma in linguaggio macchina

- Cosa fa il seguente programma?
  1. `STORE 10`
  2. `STORE 11`
  3. `RESET 12`
  4. `JMPEQ 11 12 (8)`
  5. `INC 10`
  6. `INC 12`
  7. `JMPEQ 11 11 (4)`
  8. `WRITE 10`
  9. `HALT`
  10. (*C1*)
  11. (*C2*)
  12. (*C3*)

Esempio di programma in linguaggio macchina

- Cosa fa il seguente programma?
  1. `STORE 10`
  2. `STORE 11`
  3. `RESET 12`
  4. `JMPEQ 11 12 (8)`
  5. `INC 10`
  6. `INC 12`
  7. `JMPEQ 11 11 (4)`
  8. `WRITE 10`
  9. `HALT`
  10. (*C1*)
  11. (*C2*)
  12. (*C3*)
- **Assembly** per CR (ASM_{CR})

Linguaggi di basso livello

- I primi linguaggi per i computer assomigliavano a L_{CR}
- Per semplificare la scrittura di programmi in linguaggi macchina sono stati introdotti i **linguaggi assembly**
  - Rappresentazioni simboliche di linguaggi macchina
  - Corrispondenza 1-1 istruzioni L_{CR} e ASM_{CR}
  - **Traduzione** da ASM_{CR} a L_{CR} fatta da un **programma assemblatore** (*assembler*)
- ASM_{CR} e L_{CR} sono **linguaggi di basso livello**
- Distanza dall'utente
  - È possibile una codifica più astratta dell'algoritmo?

Un linguaggio imperativo di più alto livello

- Codifica dell'algoritmo in **pseudo-codice**
  - linguaggio inventato, ma con un chiaro significato

```
1. int C1, C2, C3=0;
2. read( C1, keyboard );
3. read( C2, keyboard );
4. while( C2!=C3 )
   {
5.     C1++;
6.     C3++;
   }
7. write( screen, C1 );
```

Un linguaggio imperativo di più alto livello

- Codifica dell'algoritmo in **pseudo-codice**
  - linguaggio inventato, ma con un chiaro significato

```
1. int C1, C2, C3=0;
2. read( C1, keyboard );
3. read( C2, keyboard );
4. while( C2!=C3 )
   {
5.     C1++;
6.     C3++;
   }
7. write( screen, C1 );
```

- **Linguaggio** L_1 (indipendente da CR)

Caratteristiche di un linguaggio di alto livello

- In cosa si distingue L_1 da ASM_{CR}?
  - Minore rigidità nel formato delle istruzioni (spazi, indentazioni, a-capo, numerazione delle istruzioni ininfluente)
  - Possibilià di usare nomi simbolici (`screen`, `keyboard`)
  - Possibilià di definire variabili (`C1`, `C2`, `C3`)
  - Possibilià di specificare il tipo delle variabili (`int`)
  - Rappresentazione esplicita del costrutto di iterazione (`while`)
  - Possibilità di raggruppare istruzioni ({ `...` })
- Possibilità aggiuntiva: definire **funzioni**

Funzioni

- **Definizione** di una funzione `somma`

```
1.  int somma( int A, int B )
    {
2.      int S=0;
3.      while( B!=S )
        {
4.          A++;
5.          B++;
        }
    }
    ...
```

- **Uso** della funzione `somma`

```
11. int C1, C2=0;
12. read( C1, keyboard );
13. read( C2, keyboard );
14. write( screen, somma( C1, C2 ) );
```

Traduzione di L_1

- Per essere usato da CR, un programma in L_1 deve essere tradotto
- Varie possibilità:
  - **compilazione** di L_1 in un programma in L_{CR}
  - **compilazione** di L_1 in un programma in ASM_{CR} e poi uso dell'assembler
  - **interpretazione** di ciascuna istruzione di L_1 da parte di un programma in L_{CR}
  - traduzione (compilazione/interpretazione) in un **codice intermedio** con successiva compilazione/interpretazione

Linguaggi di più alto livello

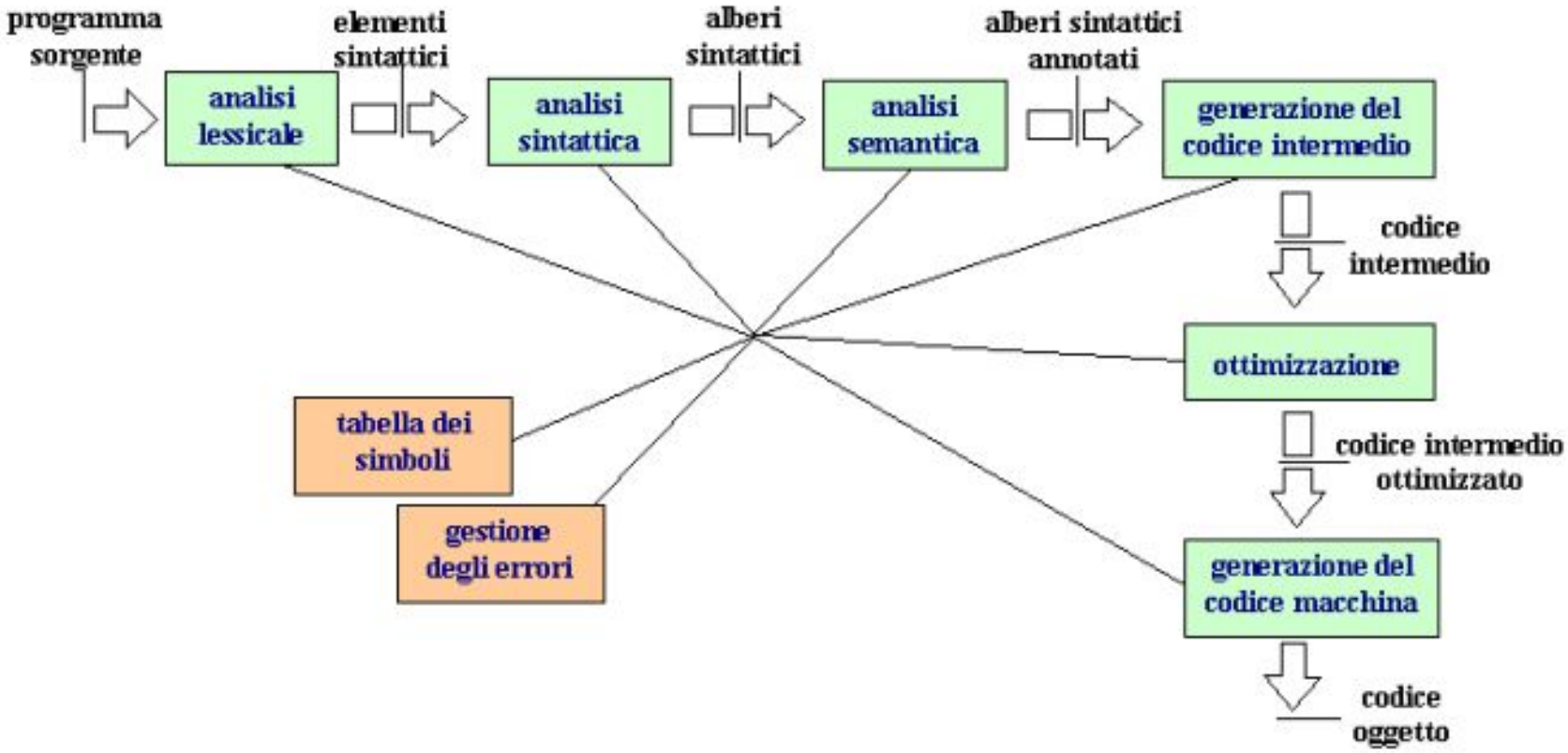
- Linguaggi di programmazione **dichiarativi**
  - Lingauggi logici e a vincoli, usati in *intelligenza artificiale*
  - Usano la logica come base per la definizione di un programma
  - Utili per ragionare sulla conoscenza
  - Usati anche in *ambito commerciale* per risolvere problemi "difficili" (ottimizzazione combinatoria)
- Linguaggi di **marcatura**
  - Usati per scrivere pagine Web
  - HTML
- Linguaggi per le **basi di dati**
  - Consentono di definire e interrogare basi di dati
  - SQL
- Spesso i linguaggi di più alto livello sono **interpretati**

Come definire un linguaggio di programmazione?

- Esistono degli elementi comuni a **tutti** i linguaggi
  - **Alfabeto**: definisce quali simboli usare per scrivere programmi
  - **Sintassi**: definisce le regole che i simboli devono rispettare per ottenere un programma corretto
  - **Semantica**: definisce il significato dei programmi

Implementazione di un linguaggio

- Implementare: definire una trasformazione (traduzione) che prende in input il codice ad alto livello, e lo rende eseguibile
- Occorre definire varie trasformazioni (passi di traduzione)



Parte VI

Sintassi dei linguaggi di programmazione

Categorie sintattiche, simboli terminali, produzioni e scopo

- Una grammatica consiste di 4 elementi:
  - $\mathcal{N}$ Insieme dei **simboli non terminali**, detti anche **categorie sintattiche**.
    - Rappresentano **categorie linguistiche**
    - Rappresentate tra parentesi angolate: `<frase>`, `<soggetto>`, `<verbo>`, `<avverbio>`, ...
  - $\mathcal{T}$ Insieme dei **simboli terminali**.
    - Sequenze di caratteri che appaiono nel linguaggio e che vanno **considerati come un tutt'uno**
    - Esempio: `Marco`, `lavora`, `bene`
  - $\mathcal{P}$ Insieme delle **regole**, dette anche **produzioni**.
    - Specificano in che modo, in una stringa, un non terminale può essere **sostituito**
  - σ Lo **scopo** della grammatica, detto anche **simbolo iniziale**
    - È un simbolo non terminale.
    - Specifica la **radice** da cui partire per applicare le produzioni.

Formato delle regole di produzione

- Nei linguaggi di programmazione il formato delle regole ha questa forma:

$$\text{simbolo n.t.} \gg \text{sequenza di simboli t. e n.t.}$$

- Grammatiche **context-free**
  - Un terminale può essere rimpiazzato indipendentemente dai simboli che lo precedono o seguono (contesto)
- Esempio: grammatica della lingua italiana.

$\mathcal{N}$: { `<frase>`, `<soggetto>`, `<verbo>`, `<avverbio>` }
$\mathcal{T}$: { `Marco`, `lavora`, `bene` }
$\mathcal{P}$: { `<frase>` $\gg$ `<soggetto>` `<verbo>` `<avverbio>` (1)
`<soggetto>` $\gg$ `Mario` (2)
`<verbo>` $\gg$ `lavora` (3)
`<avverbio>` $\gg$ `bene` } (4)
σ: `<frase>`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<frase>` $\gg$ `<soggetto> <verbo> <avverbio>`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

$\mathcal{P}_{(2)}$: `<soggetto>` $\gg$ `Mario`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

$\mathcal{P}_{(2)}$: `Mario <verbo> <avverbio>`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

$\mathcal{P}_{(2)}$: `Mario <verbo> <avverbio>`

$\mathcal{P}_{(3)}$: `<verbo>` $\gg$ `lavora`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

$\mathcal{P}_{(2)}$: `Mario <verbo> <avverbio>`

$\mathcal{P}_{(3)}$: `Mario lavora <avverbio>`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

$\mathcal{P}_{(2)}$: `Mario <verbo> <avverbio>`

$\mathcal{P}_{(3)}$: `Mario lavora <avverbio>`

$\mathcal{P}_{(4)}$: `<avverbio>` $\gg$ `bene`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

$\mathcal{P}_{(2)}$: `Mario <verbo> <avverbio>`

$\mathcal{P}_{(3)}$: `Mario lavora <avverbio>`

$\mathcal{P}_{(4)}$: `Mario lavora bene`

Derivazione

- Come facciamo a sapere che `Mario lavora bene` è una **frase corretta** della lingua italiana?
- Seguiamo un processo di **derivazione**, a partire da σ.

$$\sigma : \texttt{<frase>}$$

$\mathcal{P}_{(1)}$: `<soggetto> <verbo> <avverbio>`

$\mathcal{P}_{(2)}$: `Mario <verbo> <avverbio>`

$\mathcal{P}_{(3)}$: `Mario lavora <avverbio>`

$\mathcal{P}_{(4)}$: `Mario lavora bene`

- Le frasi **lecite** (sintatticamente corrette) sono tutte e sole quelle **derivabili** attraverso le regole della grammatica.

Linguaggi generati da grammatiche

- Possiamo definire l'insieme di stringhe derivabili da un simbolo non-terminale di una grammatica.

Definizione (linguaggio di un simbolo non terminale)

Data una grammatica, ed un simbolo non terminale di essa `<non-term>`*, diciamo* **linguaggio di** `<non-term>` *nella grammatica considerata l'insieme di stringhe di* **soli simboli terminali derivabili da** `<non-term>`*.*

- Il linguaggio di una grammatica è quello del suo scopo o simbolo iniziale (σ).

EBNF (Extended Bakus-Naur Form)

- EBNF è una **notazione** per definire una grammatica.
  - Produzioni: <simbolo n.t.> $\gg$ <sequenza di simboli>
  - **Possibile assenza** di un elemento sintattico: [...]
  - **Ripetizione** di un elemento almeno una volta: (...)$^{+}$
  - **Ripetizione** di un elemento zero o più volte: (...)*
  - **Alternativa** tra due elementi: ... | ...

Grammatiche regolari

- Sono una classe importante di grammatiche
- Esiste un algoritmo efficiente per riconoscere i **linguaggi regolari**
  - Solo due formati possibili di regole
    - `<non-term>` ≫ `term <non-term>`
    - `<non-term>` ≫ `term`
  - Esempio: numeri binari
    - `<bin>` ≫ `0 | 1 | 0 <bin> | 1 <bin> |`
- **Espressioni regolari**: una notazione alternativa delle grammatiche regolari, molto usate
  - editor per manipolare testi in base a *pattern*
    - alternativa, parentesi tonde, quadre, graffe
    - quantificazione: `?`, `*`, `+`
    - inizio/fine stringa: `^`, `$`