Programmazione

Andrea Passerini
passerini@disi.unitn.it

Informatica

Linguaggi di programmazione

- Il calcolatore è in grado di comprendere solo istruzioni in linguaggio macchina.
- Il linguaggio macchina non è adatto alla scrittura di programmi poiché troppo distante dal linguaggio naturale.
- I linguaggi di programmazione permettono di scrivere programmi più agevolmente mantenendo le caratteristiche di non ambiguità e non ridondanza necessarie.
- Per poter essere eseguito dal calcolatore, un programma scritto con un linguaggio di programmazione deve essere *tradotto* in linguaggio macchina.

Classificazione dei linguaggi

In base al livello in cui si collocano tra macchina e uomo

basso livello

- E' il linguaggio assemblatore (*assembly*)
- c'è una corrispondenza uno ad uno tra istruzioni in assembly ed istruzioni in linguaggio macchina
- è specifico per ogni CPU
- permette di sfruttare appieno le potenzialità della CPU (il suo set di istruzioni)

esempio

```
add R01,R02,R03   ;R01 ← R02 + R03
addi R02,R01,0    ;R02 ← R01 + 0
bltz R01,-2       ; se (R01 < 0) vai a PC-2
```

Classificazione dei linguaggi

In base al livello in cui si collocano tra macchina e uomo

alto livello

- E' svincolato dalla struttura della macchina
- è più agevole per il programmatore in quanto più simile al linguaggio naturale
- evita di dover conoscere le caratteristiche specifiche della CPU
- permette di utilizzare lo stesso programma su CPU diverse (con opportuni traduttori)
- per ogni CPU deve esistere il relativo traduttore del linguaggio
- Il traduttore non sempre produce codice ottimizzato

Alto livello

Esempi

- C

```
#include <stdio.h>

int main()
{
   printf("Hello, World\n");

   return 0;
}
```

- Python

```
print("Hello, World")
```

Classificazione dei linguaggi

In base alla struttura che devono avere i programmi

Programmazione strutturata

- si attiene ai concetti di diagrammi strutturati per gli algoritmi (Teorema di Böhm-Jacopini)
- introduce il concetto di *visibilità* di una variabile (*scope*) (variabili *locali* ad un blocco).
- permette maggiore riusabilità , leggibilità e manutenzione dei programmi.
- Esempi: Pascal, Ada, C, Python

Esempi: C

```
uint max(uint[] V, uint size)
{
 uint max = 0;
 uint i;

 for(i = 0; i < size; i++)
  if(V[i] > max)
   max = V[i];

 return max;
}

int main()
{
  uint[] V = {0,1,2,3,4,5};
  uint size = 6;

  uint max = max(V, size);

  printf("%d\n",max);
}
```

Esempi: Python

```
def max(V):
    max = 0
    for i in V:
        if(V[i] > max):
            max = V[i]
    return max

V = range(6)
max = max(V)
print(max)
```

Classificazione dei linguaggi

In base alla struttura che devono avere i programmi

Programmazione orientata agli oggetti

- presuppone di raggruppare dati e procedure che operano su di essi in entità (*classi*)
- una classe è dotata di *attributi* (i dati) e *metodi* (le procedure) per accedere ed operare sugli attributi
- un *oggetto* è un'istanza di una determinata classe, con specifici valori per i suoi attributi (che possono variare durante l'esecuzione)
- E' il paradigma di programmazione più usato attualmente
- Esempi: C++, Java, Smalltalk, Python

Esempi: C++

```
#include <stdio.h>

class Vector{

  int* V;
  int size;

public:

  Vector(int V_[], int size_){
    size = size_;
    V = new int[size];
    int i;
    for(i = 0; i < size; i++)
      V[i] = V_[i];
  }

  ~Vector(){
    delete[] V;
  }

  int getMax(){
    int max = 0;
    int i;
    for(i = 0; i < size; i++)
      if(V[i] > max)
       max = V[i];
    return max;
  }
};
```

```
int main()
{
   int V[] = {0,1,2,3,4,5};
   int size = 6;

   Vector Vec(V,6);
   printf("%d\n", Vec.getMax());
}
```

Esempi: Python

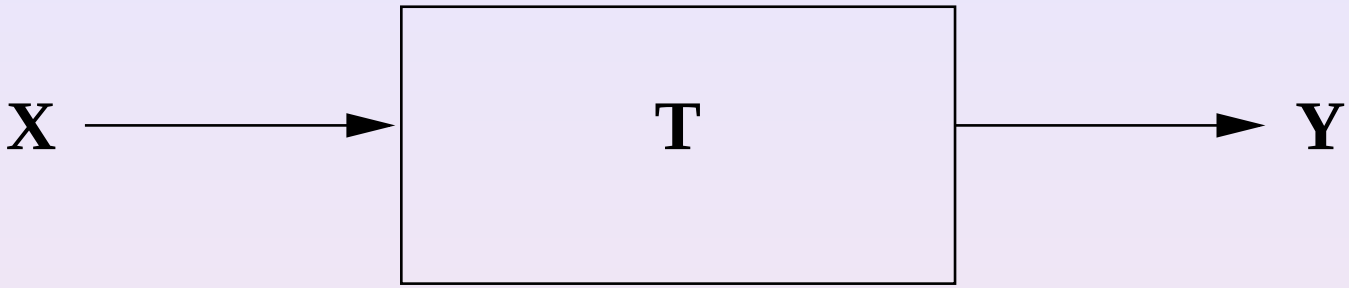
```
class Vector:

    def __init__(self, V_):
        self.V = V_[:]

    def getMax(self):
        max = 0
        for i in self.V:
            if(self.V[i] > max):
                max = self.V[i]
        return max


V = range(6)
Vec = Vector(V)
max = Vec.getMax()
print(max)
```

Traduzione



- Dati:
  - una sequenza di istruzioni *X* descritte in un linguaggio *sorgente* (e.g. C).
  - un traduttore *T* in linguaggio macchina per una certa CPU *C*
- L'esecuzione di *T* con ingresso *X* produce in uscita una sequenza di istruzioni *Y* descritte in linguaggio *eseguibile* da *C*.

Traduzione

Esistono due possibili approcci alla traduzione

Compilazione

- Il programma sorgente viene compilato nel suo rispettivo eseguibile una volta per tutte.
- La versione eseguibile viene utilizzata ogni volta si debba eseguire il programma.
- Esempi di linguaggi compilati: Pascal, C

Esempio: C

- Compilazione del file `max.c` nell'eseguibile `max`:

```
gcc -o max max.c
```

- Esecuzione dell'eseguibile `max`

```
./max
```

Traduzione

Interpretazione

- La traduzione in instruzioni in linguaggio macchina avviene solo al momento dell'esecuzione del programma.
- Il programma non viene eseguito direttamente dalla CPU ma tramite un *interprete* che traduce e manda in esecuzione le istruzioni del programma una per una.
- Esempi di linguaggi interpretati: Perl, PHP, Python, bash

Traduzione

Esempio: Python

- Esecuzione del programma `max.py` tramite l'interprete `python`:

```
python max.py
```

- Esecuzione dell'interprete `python` in modalità interattiva:

```
> python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| ...
[GCC 4.2.1 Compatible Apple LLVM 6.0 ...
Type "help", "copyright", "credits" ...
>>> print("Hello, world")
Hello, world
>>>
```

Compilatori vs Interpreti

tempi di esecuzione L'interprete deve tradurre al volo ogni singola istruzione (impiegando tempo) e non può ottimizzare il codice

qualità della traduzione il compilatore, eseguendo la traduzione al di fuori dell'esecuzione del programma, ha tutto il tempo di ottimizzare la traduzione realizzando codice più efficiente

portabilità del codice un programma eseguibile non può funzionare su una CPU diversa da quella per cui è stato compilato

Compilatori vs Interpreti

modificabilità del codice un programma eseguibile non può essere modificato, e recuperare il sorgente a partire dall'eseguibile è estremamente difficile (*reverse code engineering*)

scrittura dei programmi la scrittura di software tipicamente comporta modifiche successive al codice sorgente via via che viene scritto. In un linguaggio compilato qualunque modifica al programma comporta la ricompilazione del tutto.

interattività nello sviluppo un interprete può eseguire codice in maniera interattiva, facilitando notevolmente la programmazione.

Compilatore + *Virtual Machine*

- è una soluzione ibrida tra compilazione ed intepretazione
- Il codice sorgente viene compilato in un linguaggio intermedio (*bytecode*) (una volta per tutte)
- il bytecode viene interpretato da una *virtual machine* (VM) ossia un interprete di basso livello che emula una CPU.
- vantaggi
  - Le ottimizzazioni più onerose vengono fatte durante la prima compilazione
  - Il bytecode è non modificabile
  - Lo stesso bytecode può essere eseguito su tutte le CPU per cui esiste un'implementazione della VM
  - E' più facile scrivere VM per CPU diverse che compilatori
- svantaggi
  - Il bytecode deve comunque essere convertito in codice macchina a runtime $\rightarrow$ più lento di eseguire direttamente codice macchina