Evoluzione dei linguaggi di programmazione

Programmazione
Corso di laurea in Comunicazione digitale

La traduzione dei linguaggi

- Evoluzione verso sistemi di codici complessi e potenti, orientati più all'uomo che alla macchina
- La sfida degli anni ’50 sulla traduzione dei linguaggi
- Linguaggi ad ***alto livello*** e a ***basso livello*** ovvero *i linguaggi macchina*

I linguaggi anni ‘50

- I primi linguaggi ad alto livello
  - **FORTRAN**, introduce il concetto di sottoprogrammi che operano su dati comuni
  - **ALGOL**, introduce il concetto di struttura dei programmi e di procedure *ricorsive*, ovvero che richiamano sè stesse
  - **COBOL**, introduce il concetto di FILE e di descrizione dei dati

I linguaggi anni ‘60

- Notazioni per la descrizione dei linguaggi
- Meno enfasi sull’efficienza, attenzione al modello di computazione
- I linguaggi orientati al problema
  - **LISP**, uniformità tra dati e programmi e un paradigma di programmazione basato sul concetto di funzione
  - **APL**, linguaggio matematico, ricco di notazioni e operatori per operare su strutture come vettori e matrici
  - **SNOBOL**, offre strumenti utili per la manipolazione di sequenze di caratteri

Evoluzione dei linguaggi – ‘70

- Metodologia di programmazione
  - **PASCAL**, ha lo scopo di insegnare la programmazione strutturata, una possibile risposta alla necessità di programmare con un metodo. Segue **Modula-2** che introduce il concetto di modulo.
  - **C**, linguaggio ad alto livello con visibilità e accesso alla macchina
  - **Prolog**, linguaggio basato sulla logica, non convenzionale, diventato di nicchia

I linguaggi anni ‘80

- Programmazione in grande, esigenza di modularità e di astrazione
  - La programmazione a oggetti. Una classe definisce un insieme di oggetti e le procedure per manipolarli
  - **SmallTalk**, **Objective-C**, **C++**

I paradigmi di programmazione

- Forniscono la filosofia e la metodologia con cui si scrivono i programmi
- I linguaggi devono *consentire* ma soprattutto *favorire* l'adozione di un paradigma
  - Procedurale
  - Funzionale
  - Modulare
  - Orientato agli oggetti

Paradigma procedurale

- Enfasi sulla soluzione algoritmica dei problemi mediante modifica ***progressiva*** dei dati in memoria
  - Esecuzione sequenziale di istruzioni
  - Cambiamento dello stato di memoria (le variabili) tramite assegnamento
  - Programmazione per effetto collaterale (*side-effect*)
- Aderenti al modello della macchina di von Neumann
- Molto efficienti
- Limitato nello sviluppo e mantenimento di sw complessi
- I linguaggi **imperativi**: **Pascal**, **C**

Influenza del modello di macchina

- **Concetto di istruzione**
  - l'unità di base del programma, memorizzata in successive celle di memoria
- **Concetto di sequenzialità e iterazione**
  - Il programma assolve il compito eseguendo le istruzioni in sequenza
  - Presente in diversi costrutti dei linguaggi e in tutto il processo di esecuzione
- **Concetto di variabile e di assegnamento**
  - Le celle di memoria hanno un indirizzo e contengono i dati da manipolare
  - Le variabili hanno un nome e un valore
  - L'assegnamento di un valore a una variabile equivale al trasferimento di un dato in una cella

È sorprendente che il computer di Von Neumann sia rimasto così a lungo il paradigma fondamentale dell'architettura dei calcolatori.

Ma dato il fatto, non è sorprendente che i linguaggi imperativi siano i principali oggetti di studio e sviluppo.

Perchè come Backus ha sottolineato i linguaggi imperativi hanno solide radici nell'architettura della macchina di Von Neumann e ne sono l'immagine.

Horowitz *Fundamentals of Programming Languages*, 1983

La macchina di Von Neumann

- Il calcolatore dello I.A.S. (Princeton, 1952): Von Neumann, Goldstein, Burks …
- Primo modello con programmazione e memorizzazione del programma
  - Organo aritmetico-logico (oggi CPU o data-path)
  - Memoria
  - Organo di controllo
  - Organo per la gestione input/output

Paradigma funzionale

- Primo tentativo di non rifarsi al modello di macchina di von Neumann
- La computazione avviene tramite funzioni che applicate ai dati riportani nuovi valori
  - Ogni funzione è un modulo a sé, dipendente unicamente dal valore dei suoi argomenti
  - L'effetto globale è ottenuto concatenando opportunamente funzioni anche richiamando sé stesse (*ricorsione*)
  - Modello che si rifà alla teoria delle funzioni ricorsive
  - Scarso supporto ai costrutti di ripetizione tramite iterazione
- **Lisp**, **ML**

Componenti dei linguaggi funzionali

- Un insieme di funzioni primitive
- Una legge di **composizione** di funzioni
- Una legge di **applicazione** di una funzione ai suoi argomenti
- Un insieme di oggetti su cui operare

Paradigma modulare

- Introduce il concetto di modulo che nasconde i dati all'utente
- I dati possono essere letti solo tramite un'opportuna interfaccia
- **Modula-2**, **Ada**

Esempi

- Il problema di sommare i numeri dispari in un insieme dato
  - In **Pascal** **sm_dispari_pascal**
  - In **APL** **sm_dispari_apl**
  - In **Lisp** **sm_dispari_lisp**
  - In **Logo** **sm_dispari_logo**
  - In **Forth** **sm_dispari_forth**

I file sono raggiungibili dalla pagina:
http://homes.dico.unimi.it/~alberti/Prog09/Lucidi/Intro/ling.html

Pascal

```
program sommaNumeriDispari (output);
type     indTermini = 1..100;
         vetTermini = array [indTermini] of integer;
var      termini : vetTermini;

   function sommaDispari (n: indTermini; term: vetTermini): integer;
   {input: n, la lunghezza dell'array term; e term array numeri}
   {output: la somma dei numeri dispari contenuti in term}
        var i: indTermini;
        somma: integer;
   begin
     somma:= 0;
     for i:= 1 to n do
       if odd(term[i]) then
          somma:= somma + term[i];
     sommaDispari:= somma;
   end;
begin
  termini[1]:= 23; termini[2]:= 34; termini[3]:= 7; termini[4]:= 9;
  writeln (sommaDispari (4, termini);
end.
```


Lisp

Definizione della funzione sommadispari:

```
(defun sommadispari (termini)
(cond ((null termini) 0)
      ((oddp (car termini)) (+ (car termini)
                               (sommadispari (cdr termini))))
(t (sommadispari (cdr termini)))))
```

L'istruzione che esegue la funzione:

```
(sommadispari '(23 34 7 9))
```

Il processo di esecuzione:

```
+ 23 (sommadispari '(34 7 9))
     (sommadispari '(7 9))
     (+ 7 (sommadispari '(9))
          (+ 9 (sommadispari '())
               0
          9
     16
```


APL

Definizione della funzione sommadispari:

```
somma ← sommadispari termini
[1] somma ← +/(2|termini)/termini
```

L'istruzione che esegue la funzione:

```
sommadispari 23 34 7 9
```

Il processo di esecuzione:

```
termini ← 23 34 7 9
(2|termini) ← 1 0 1 1   esegue modulo 2
(2|termini)/termini← 23 7 9   compressione vettori
+/(2|termini)/termini ← 39   riduzione con somma
```

FORTH

Definizione della funzione sommadispari:

```
sommadispari
    0
    do
         swap dup 2 mod
         if +
         else drop
         then
    loop
```

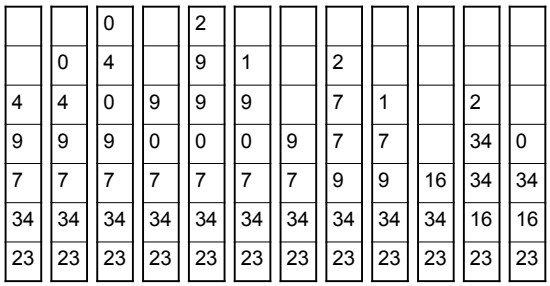
L'istruzione che esegue la funzione:

```
23 34 7 9 4 sommadispari
```


Esecuzione in FORTH

- Forth gestisce uno stack su cui carica I dati prima di esguire il codice
- `swap` scambia il contenuto delle prime due posizioni dello stack
- `dup` duplica il contenuto della cima dello stack e lo colloca sullo stack
- per ogni numero nel codice si sottointende che si effettui una operazione di push sullo stack
- `if` effettua un test sulla cima dello stack se vale
  - 1 allora esegue le istruzioni tra `if` e `else` (+ nel nostro caso)
  - 0 allora esegue da `else` a `then` (drop nel nostro caso)
- alla fine del processo sullo stack ci sara' la somma dei numeri dispari.

Il processo d'esecuzione - 1



Il processo d'esecuzione - 2