Emerging Programming Paradigms

Cenni di storia dei linguaggi di programmazione

Luca Padovani

Dipartimento di Informatica – Scienza e Ingegneria, Università di Bologna

paradigmi di programmazione

Paradigma	**Programma**	**Linguaggi**
Imperativo	sequenza di azioni	C, Pascal, C++, Java, Scala, Python, Scheme
Object-oriented	oggetti che comunicano	Smalltalk, C++, OCaml, Java, Scala, Python
Funzionale	espressione	Haskell, OCaml, Scala, Scheme, F#

Nota

- quasi tutti i linguaggi in uso sono **multi-paradigma**

anni 50

- **Assembly**: corrispondenza 1-1 con codice macchina, mancano visioni su linguaggi ad alto livello, si teme che non sarà mai possibile scrivere programmi complessi
- **FORTRAN**: linguaggio per **calcoli numerici**, introduce **procedure** (no ricorsione), cicli, **espressioni** in notazione infissa, I/O formattato, usato ancora oggi (con raffinamenti), richiede 10 anni/uomo di sviluppo, bootstrap difficilissimo (gli studi contemporanei di Chomsky furono applicati solo in seguito)
- **COBOL**: linguaggio per gestione aziendale, introduce il **record**

anni 60

- **ALGOL**: grammatica in forma **BNF** (Backus-Naur Form), primo linguaggio indipendente dall'architettura, introduce **blocchi** con **variabili locali**, **ricorsione**
- **BASIC**: "programmazione per tutti", sarà incorporato nei primi Personal Computer (IBM e Apple) 20 anni dopo, facile da imparare ma poco strutturato (`goto`)
- **Simula 67**: astrazioni di più alto livello, introduce **classi**, **oggetti** e **ereditarietà**

anni 70

- **Pascal**: programmazione **strutturata**, ruolo importante dei **tipi** per evitare errori di programmazione, ideale per imparare a programmare
- **Smalltalk**: programmazione a **oggetti** estrema, tutto è un oggetto (numeri, classi, ecc.), l'unica operazione possibile è l'invio di un **messaggio** (metodo)
- **C**: linguaggi di medio livello per facilitare l'accesso all'hardware e implementazione di sistemi operativi (UNIX)

anni 80

- **Ada**: sviluppato dal ministero della difesa USA, evoluzione del Pascal con **concorrenza** e **tipi astratti** (rappresentazione privata + interfaccia pubblica)
- **C++**: estensione di C con classi e oggetti, introduce i **template** (classi generiche ottenute per espansione) e **overloading** di operatori e metodi
- **PostScript**: linguaggio stack-based creato da Adobe per descrivere documenti, interprete in stampanti, verrà semplificato e reso efficiente in PDF
- **Perl**: prestazioni PC rendono possibile uso di linguaggi **interpretati**, Perl è per "sistemisti", manipolazione di file di configurazione del S.O., find/replace con **espressioni regolari**, linguaggio **write-only** (programmi difficili da leggere)

anni 90

- **Python**: linguaggio di scripting inventato da uno studente per noia durante le vacanze natalizie
- **Java**: versione ripulita e robusta di C++ per la programmazione con classi e oggetti, idea “compile-once-run-everywhere” grazie alla **JVM**, inizialmente pensato per **dispositivi mobili**, sistemi embedded (lavatrici) e Web (**applet**), diventerà mainstream
- **PHP**: produzione di pagine **Web** da database
- **JavaScript**: linguaggio di scripting per la produzione di **pagine Web interattive** (gestione click e altri eventi, animazioni, effetti grafici)
- **Erlang**: linguaggio per programmazione concorrente estrema (“tutto è un processo”) e resiliente (“compile-once-run-forever”)

anni (20)00

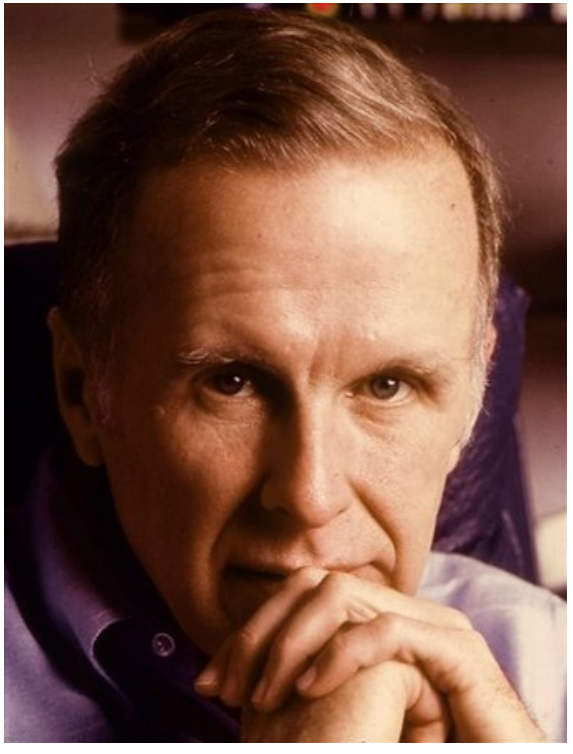
- **C#**: vuole essere C++ “fatto bene”, compilatore per la piattaforma .NET di Microsoft e successivamente altri S.O. (mono), contributi da un comitato di esperti
- **Scala**: linguaggio a oggetti con sintassi “flessibile”, compilatore per JVM
- **Go**: linguaggio di Google per programmazione **concorrente** e **distribuita** su **scala globale**, comunicazione su **canali**, elimina **classi** a favore delle **interfacce**

1977, colpo di scena

FP (1978)

John Backus (1924–2007) vince il premio Turing per il **FORTRAN** nel 1977, ma durante la lezione d'onore critica aspramente il paradigma imperativo e introduce **FP** (Functional Programming)

> "Conventional programming languages are growing ever more enormous, but **not stronger**. Inherent defects at the most basic level cause them to be both **fat and weak**: their primitive **word-at-a-time style of programming** [...], their **division of programming into a world of expressions and a world of statements**, their **inability to effectively use powerful combining forms** for building new programs from existing ones, and their **lack of useful mathematical properties** for reasoning about programs."



- John Backus, **Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs**, 1978

λ calcolo (anni 30)

- Alonzo Church (1903–1995) inventa un linguaggio per “calcolare le funzioni”
- tutto è una funzione (numeri, liste, costrutti di controllo, ecc.)
- funzioni di **prima classe** e di **ordine superiore**
- funzioni **anonime** es. $\lambda x.x$

Fattoriale (provare per credere)

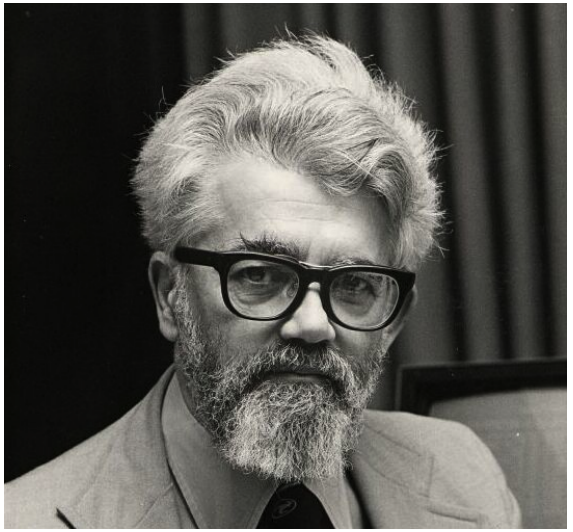
```
\n.n (\f.\a.\x.f ((\x.\y.\z.x (y z)) a x)
((\n.\f.\x.f (n f x)) x)) (\x.\y.x)
(\f.\x.f x) (\f.\x.f x)
```

Concatenazione di liste

```
(\g.(\x.g (x x)) (\x.g (x x)))
(\g.\c.\x.(\p.p (\x.\y.\x.\y.y)) x c
(g ((\n.\f.\x.f (n f x)) c) ((\p.p \x.\y.y) x)))
(\f.\x.x)
```

LISP (anni 50)

- John McCarthy (1927-2011, premio Turing 1971) crea il **LISP**, un linguaggio per l'elaborazione dell'informazione non-numerica e simbolica
- Ruolo chiave delle **liste** (LISt Processor)
- primo **garbage collector**
- notazione prefissa



```
(defun factorial (n)
  (if (= n 0)
   1
   (* n (factorial (- n 1)))))

(defun append (l1 l2)
  (if (null l1)
   l2
   (cons (first l1) (append (rest l1) l2))))
```

ML (anni 80)

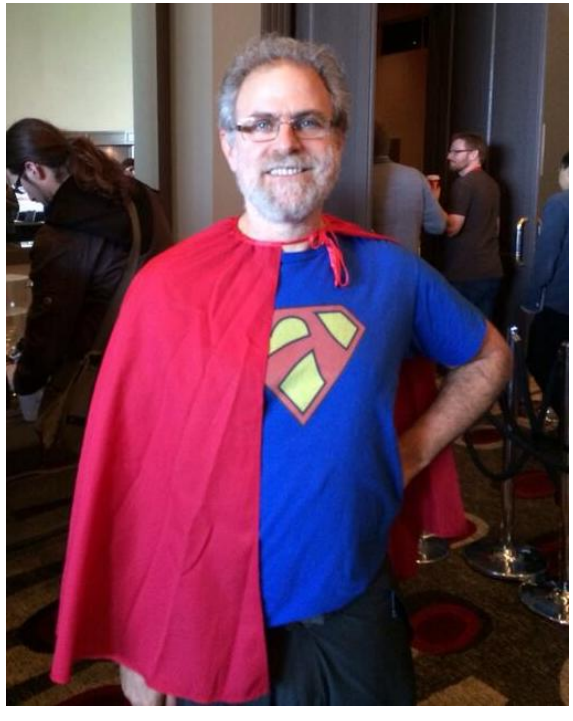
- Robin Milner (1934-2010, premio Turing 1991) crea **ML** (Meta Language) che diventerà il capostipite della più grande famiglia di linguaggi funzionali moderni
- “well-typed programs don’t go wrong”
- **type inference** e **polimorfismo parametrico**
- **pattern matching**

```
fun factorial n =
  if n = 0 then 1
  else n * factorial (n - 1)

fun append (xs, ys) =
  case xs of
    []          => ys
  | (hd :: tl) => hd :: append (tl, ys)
```

Haskell (anni 90)

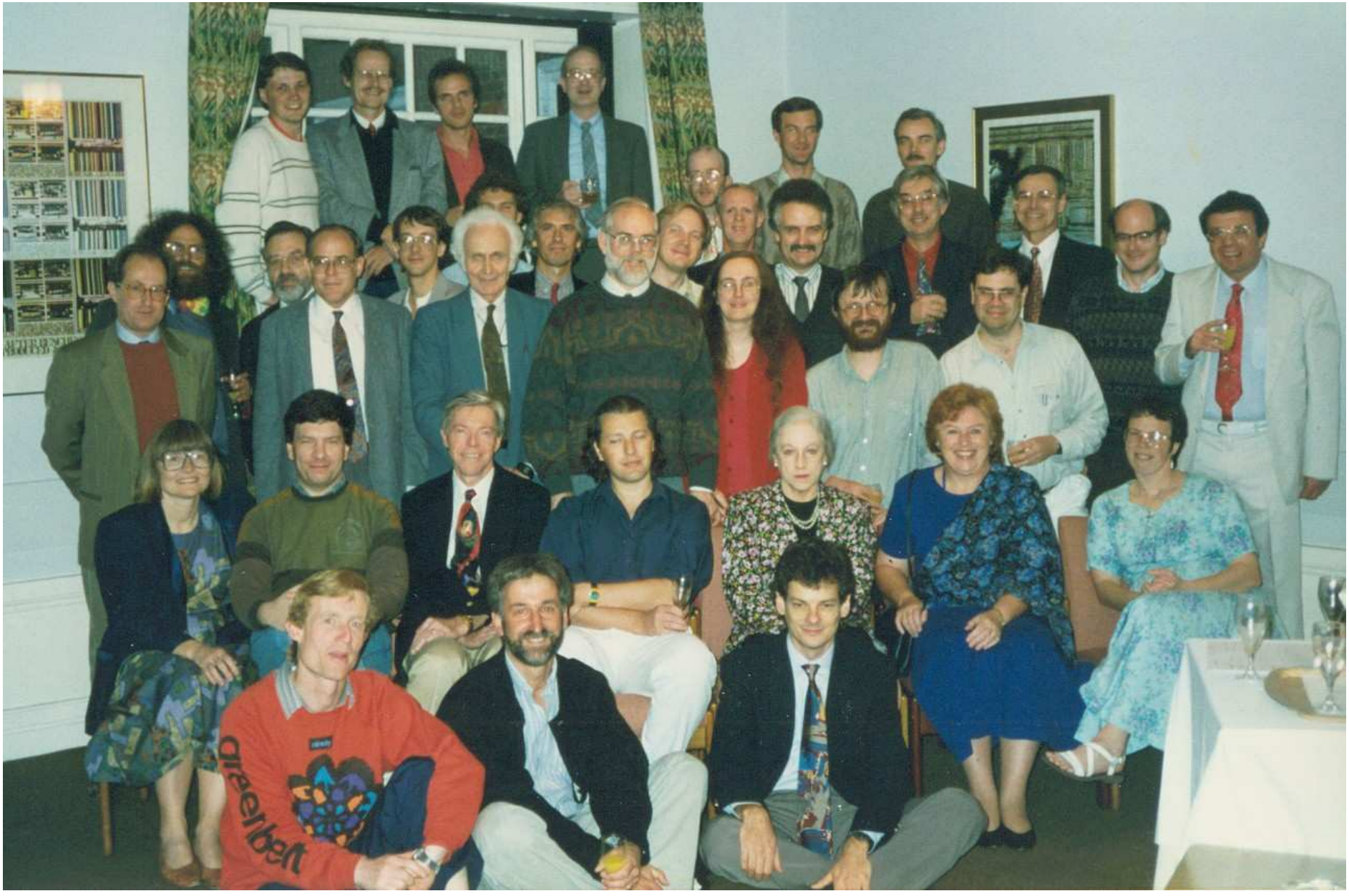
- Simon Peyton-Jones, Phil Wadler e molti altri (comitato scientifico)
- linguaggio **lazy**
- separazione tra parte pura e parte impura per mezzo di **monadi** (I/O, stato, eccezioni, non-determinismo, …)
- overloading con **type classes**



```
factorial 0 = 1
factorial n = n * factorial (n - 1)

append []       ys = ys
append (x : xs) ys = x : append xs ys
```

IFIP 1992



esperimento del Naval Surface Warfare Center (NSWC)

- Nel 1993 l'NSWC effettua uno studio sulla **rapidità di sviluppo** di prototipi in vari linguaggi di programmazione
- Haskell si dimostra essere **conciso** e **molto efficace**, anche per i neofiti (paradossalmente non verrà scelto)
- Meno codice = linguaggio di programmazione di più alto livello

Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

- Paul Hudak, Mark P. Jones, **Haskell vs. Ada vs. C++ vs. Awk vs. ...An Experiment in Software Prototyping Productivity**, 1994

moltiplicazione di matrici in Java

```
int[][] product(int[][] a, int[][] b) {
  int[][] r = new int[a.length][b[0].length];
  for (int i = 0; i < a.length; i++)
    for (int j = 0; j < b[0].length; j++)
      for (int k = 0; k < a[0].length; k++)
        r[i][j] += a[i][k] * b[k][j];
  return r;
}
```

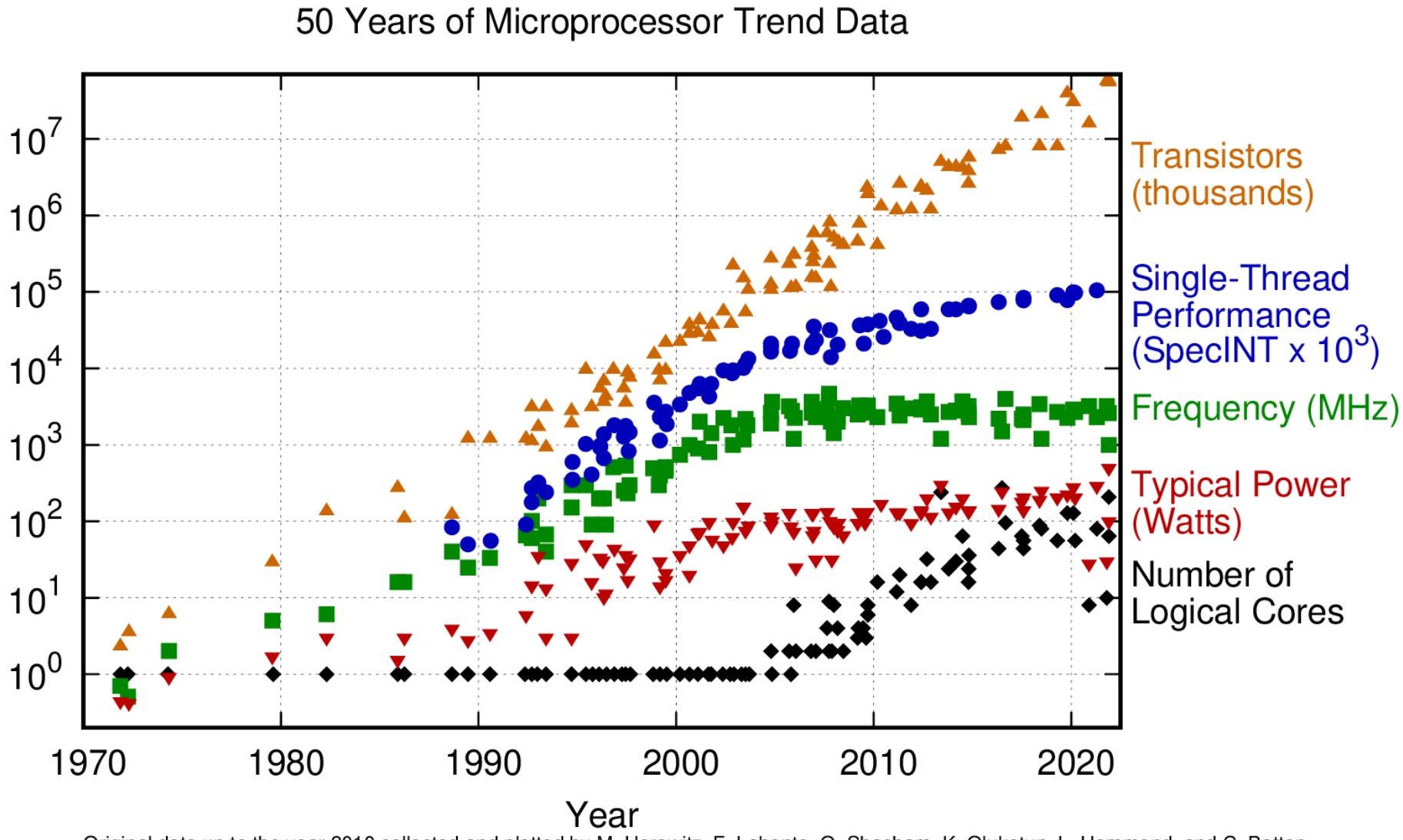
- esempio di **word-at-a-time-programming**, inizializzazione, confronto e modifica degli indici, accumulo dei prodotti
- l'unico ciclo disponibile è il `for`, quello primitivo
- codice ricco di **assegnamenti**

moltiplicazione di matrici in Haskell

```
(·) v w = sum (zipWith (*) v w)
(×) a b = map (\r -> map (· r) (columns b)) (rows a)
```

- esempio di **wholemeal programming**, es. “moltiplica ogni riga `r` di `a` con una colonna di `b`”
- `sum`, `zipWith`, `map` sono funzioni già definite (ma **non primitive**) nella libreria standard del linguaggio
- le forme di **iterazione**, **riduzione**, **composizione** di liste (e altre strutture dati) sono **programmabili**, **generiche**, **utili in molti contesti diversi**
- non ci sono **assegnamenti**!

2004, finisce la pacchia



- Herb Sutter, **A Fundamental Turn Toward Concurrency in Software**, 2005

che cosa succede?

Fino al 2004

- Per raddoppiare la velocità di esecuzione dei programmi bastava **aspettare** ∼18 mesi, più o meno il tempo in cui i produttori riuscivano a raddoppiare la **velocità dei microprocessori** grazie al progresso tecnologico

Nel 2004

- La velocità dei microprocessori si attesta attorno ai ∼3GHz, andando oltre si incontrano **limiti fisici** difficili da aggirare
  - troppa energia per funzionare
  - troppo calore da dissipare

Dopo il 2004

- Il **numero di transistor** continua ad aumentare seguendo grosso modo la legge di Moore (raddoppio di densità ogni ∼18 mesi), ma invece di investire i transistor in più per realizzare un **singolo core** molto veloce, si realizzano **tanti core** alla velocità massima possibile (∼3GHz)

conseguenze della “rivoluzione multicore”

- Per sfruttare efficacemente la disponibilità di tanti core occorre scrivere programmi in cui attività indipendenti possano essere **eseguite in parallelo su core diversi**
- Questo è **facile a dirsi ma difficile (o impossibile) a farsi**, di certo non basta più aspettare ∼18 mesi come succedeva fino al 2004
- Ad aggravare lo scenario, molti linguaggi di programmazione sono mal equipaggiati per la scrittura di programmi paralleli e concorrenti
  - i side effect (assegnamenti) sono spesso causa di conflitti tra attività
  - lo stile “word-at-a-time programming” rende difficile l’identificazione di attività complesse da eseguire in parallelo
- L’assenza di side effect e **wholemeal programming** sono proprio tra le caratteristiche fondamentali della **programmazione funzionale**

c’è un futuro (più) funzionale?

- Il paradigma di programmazione funzionale ha grande potenziale nello sviluppo di programmi, tuttavia i linguaggi funzionali sono secondari nell’industria, con poche eccezioni (es. Erlang)
- Molti linguaggi di programmazione imperativi e a oggetti (C++, Java, C#, JavaScript, Python, Ruby, …) hanno da tempo inglobato caratteristiche e costrutti tipici del paradigma funzionale

Letture consigliate

- Phil Wadler, **Why No One Uses Functional Languages**, 1989
- John Hughes, **Why Functional Programming Matters**, 1990