Introduzione ai paradigmi di programmazione e breve storia dei linguaggi funzionali

Luca Padovani

Linguaggi e Paradigmi di Programmazione

Questo documento è distribuito con licenza CC BY-NC-SA 4.0. Generato il 18 agosto 2020.

paradigmi di programmazione

Paradigma	**Un programma è**	**Esempi**
Imperativo	sequenza di azioni esecuzione ⇒ nuovo stato	C, Pascal
Object-oriented	oggetti che comunicano interazione ⇒ nuovo stato	Smalltalk
Funzionale	espressione valutazione ⇒ risultato	Haskell

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

paradigmi di programmazione

Paradigma	**Un programma è**	**Esempi**
Imperativo	sequenza di azioni esecuzione ⇒ nuovo stato	C, Pascal, C++, Java, Scala, Python, Scheme
Object-oriented	oggetti che comunicano interazione ⇒ nuovo stato	Smalltalk, C++, OCaml, Java, Scala, Python
Funzionale	espressione valutazione ⇒ risultato	Haskell, OCaml, Scala, Erlang, Scheme

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

storia (parziale) dei linguaggi di programmazione

’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**
’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**
’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: linguaggio di medio livello per facilitare l’accesso all’hardware e implementazione di sistemi operativi (UNIX)

storia (parziale) dei linguaggi di programmazione

’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)
’90	Python: linguaggio di scripting “della domenica”, versione a oggetti di Perl Java: versione ripulita e robusta di C++ per la programmazione con classi e oggetti, idea “compile-once-run-everywhere” grazie alla **JVM**, pensato per **dispositivi mobili**, sistemi embedded (lavatrici) e Web (**applet**), diventerà mainstream PHP: produzione di pagine **Web** da database JavaScript: produzione di **pagine Web interattive**
’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”, incorpora caratteristiche da linguaggi funzionali, compilatore per JVM Go: linguaggio di Google per programmazione **concorrente** e **distribuita** su **scala globale**, comunicazione su **canali**, elimina **classi** a favore delle **interfacce**

nel frattempo...

λ calcolo

anni 30

- Alonzo Church (1903–1995)
- “calcolare” con le funzioni, così come si calcola con i numeri
- tutto è una funzione (numeri, liste, costrutti di controllo, ecc.)
- funzioni **anonime**. E.g. identità

$$\lambda x.x$$

- funzioni a un argomento, **currying** (da Haskell Curry)
- funzioni di **ordine superiore** (funzioni applicate a funzioni)
- il λ nasce da errore tipografico

Fattoriale

```
λn.n (λf.λa.λx.f ((λx.λy.λz.x (y z)) a x)
((λn.λf.λx.f (n f x)) x)) (λx.λy.x)
(λf.λx.f x) (λf.λx.f x)
```

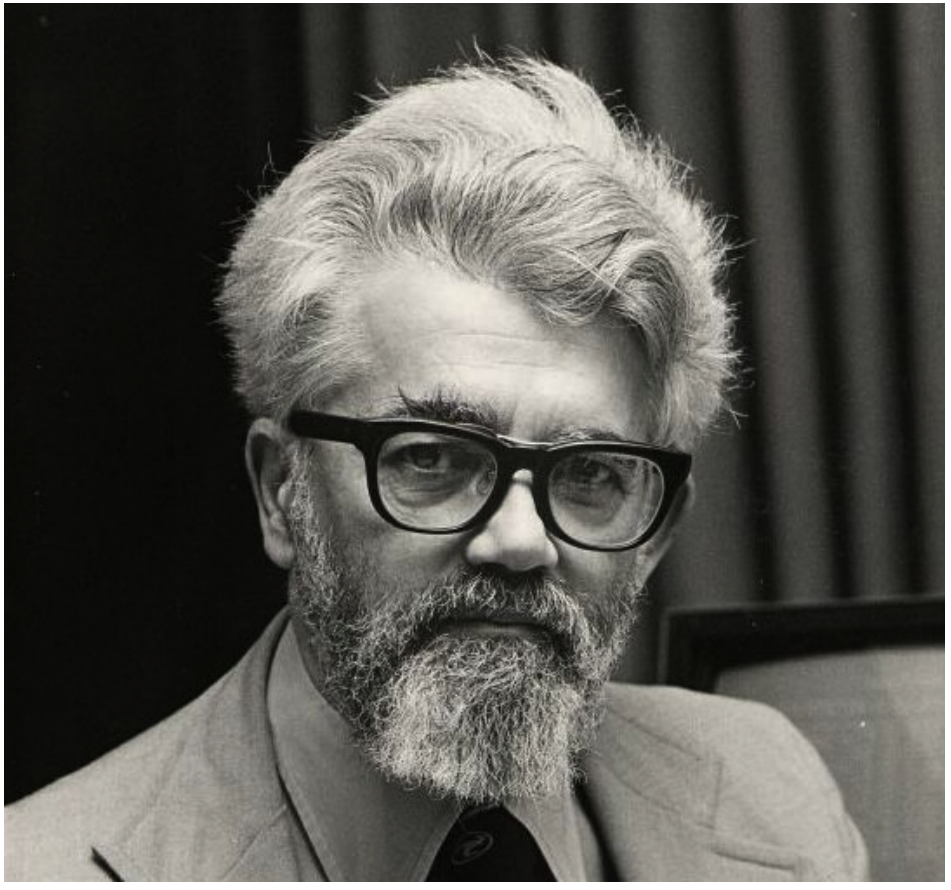
Concatenazione di due liste

```
(λg.(λx.g (x x)) (λx.g (x x)))
(λg.λc.λx.(λp.p (λx.λy.λx.λy.y)) x c
(g ((λn.λf.λx.f (n f x)) c) ((λp.p λx.λy.y) x)))
(λf.λx.x)
```

LISP

anni 50

- John McCarthy (1927–2011)
- elaborazione informazione non-numerica/simbolica, i programmi sono (anche) dati
- ruolo chiave delle **liste**
  - LISP = LISt Processor
  - “cons” e “map” nascono qui
- primo **garbage collector**
- **funzioni anonime** (λ calcolo)
- notazione prefissa (facile implementare parser)
- **impuro**, forte legame con architettura di Von Neumann



Fattoriale

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

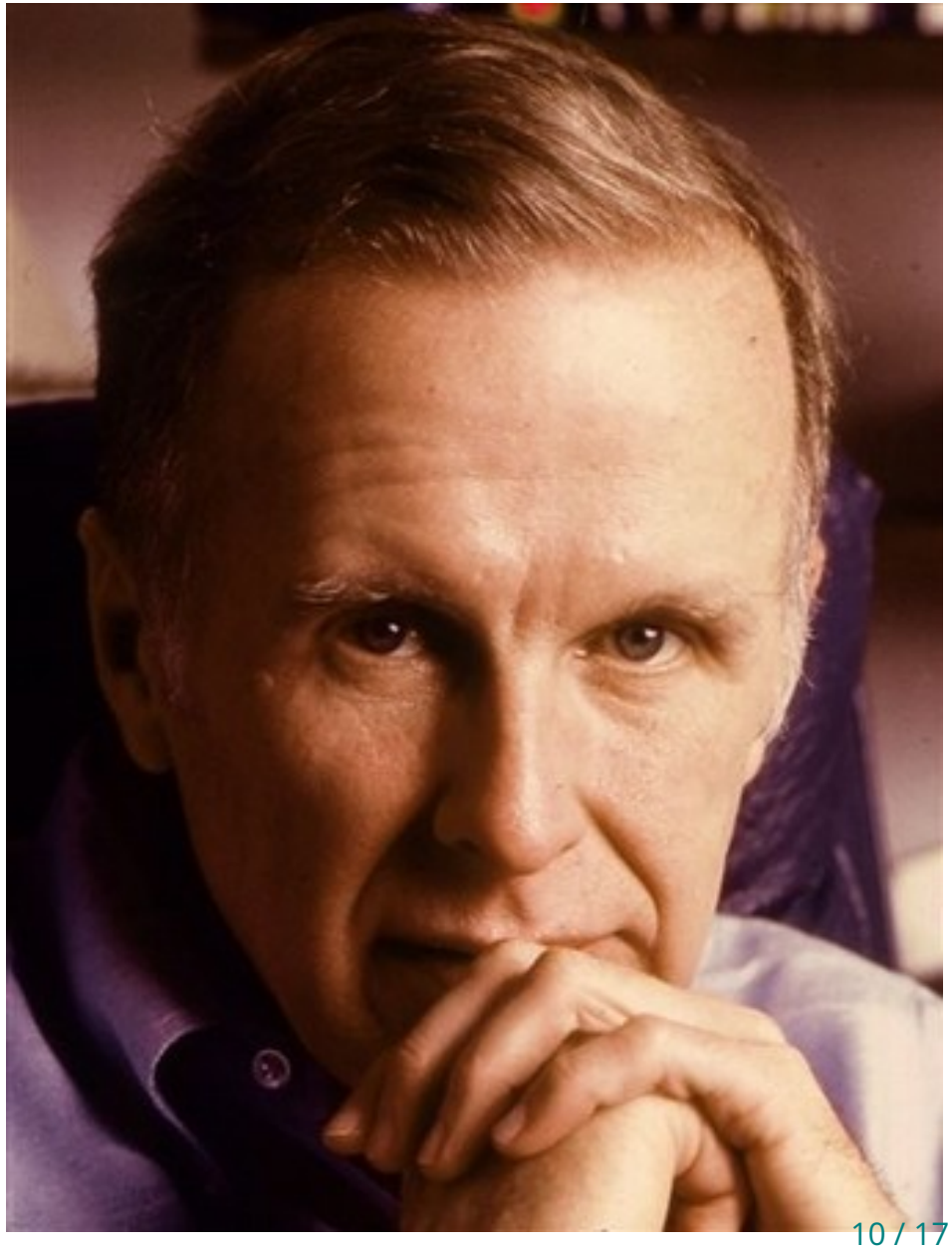
Concatenazione di due liste

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

- John Backus (1924–2007)
- vince il premio Turing per il **FORTRAN** nel 1977 ma durante la lezione d'onore critica il paradigma imperativo ("word-at-a-time" programming) e introduce **FP**
- accende l'interesse per i linguaggi funzionali, anche se FP non avrà mai successo
- programmi in stile **point-free**

$$(/+) \circ (\alpha\times) \circ \textit{Transpose}$$

prodotto scalare di 2 vettori



ML

anni 70

- Robin Milner (1934–2010)
- “spin-off” di LCF (Logic for Computable Functions), diventa linguaggio a sé stante
- linguaggio fortemente tipato (no errori di tipo a runtime)
- **type inference** (tipi inferiti dal compilatore)
- **polimorfismo parametrico** (30 anni prima dei Java generics)
- **tipi** definibili dal programmatore
- **pattern matching**
- sistema di **moduli**

Fattoriale

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

Concatenazione di due liste

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

SASL, KRC e Miranda

anni 70-80

- David Turner e altri
- linguaggi **lazy**, con valutazione “on demand” degli argomenti
- SASL introduce **guardie** e **currying**
- KRC introduce il costrutto precursore delle **list comprehension**

$$[x^2 \mid x \leftarrow [1..100], odd(x)]$$

- Miranda introduce le **sezioni**

$$(/\ 2)$$

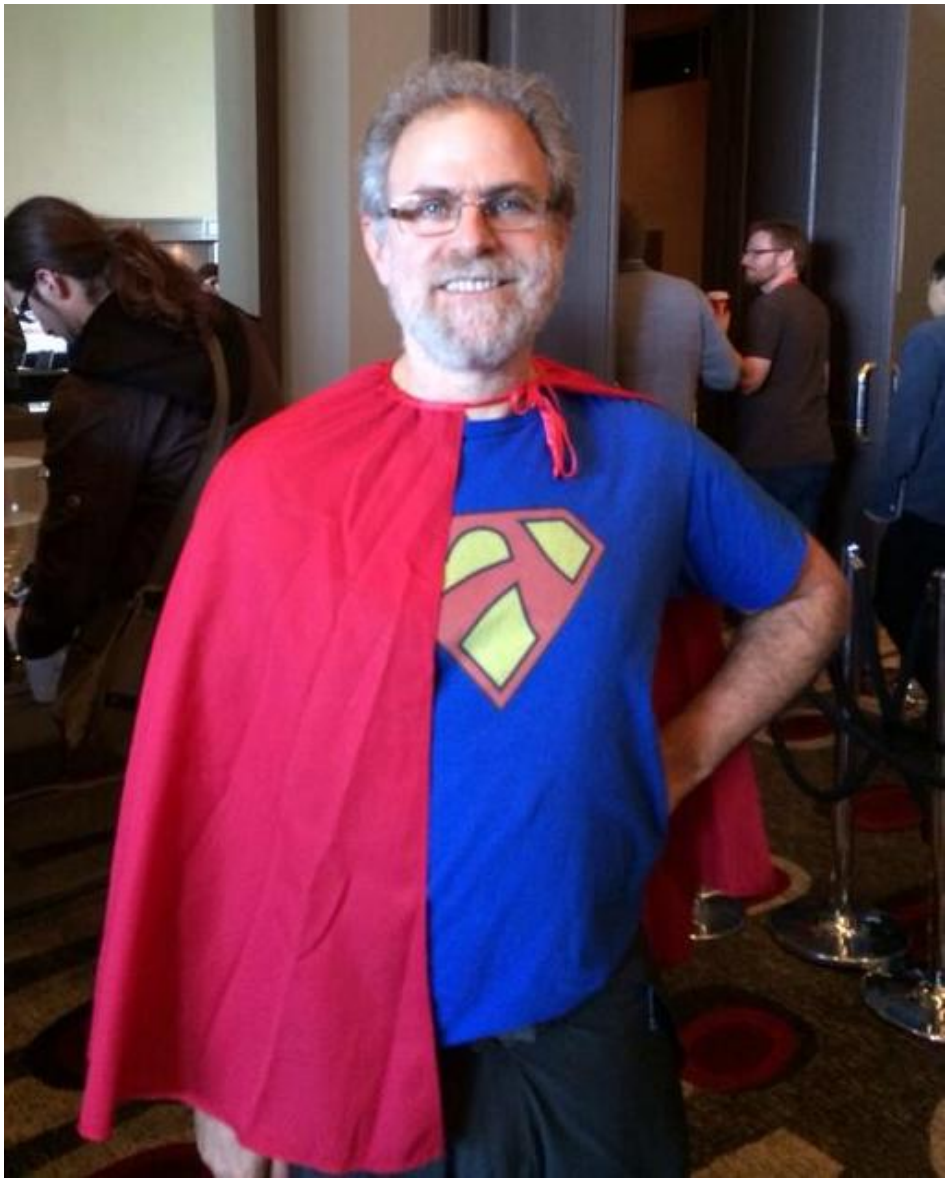
Fattoriale con guardie

```
fac n = 1,                    n = 0
      = n * fac (n - 1),      n > 0
```

Haskell

anni 90

- Phil Wadler e molti altri (comitato scientifico)
- linguaggio **lazy** “standardizzato”
- separazione tra parte pura e parte impura per mezzo di **monadi** (I/O, stato, . . .)
- overloading con **type classes**
- laboratorio di ricerca per linguaggi lazy e **sistemi di tipo** avanzati
- seconda vita dopo il 2005 per impatto su **calcolo parallelo**



Fattoriale

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

Concatenazione di due liste

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

IFIP 1992