

_FRONTEND EDITION

// CATANIA

28.06.2024

FOUR POINTS
BY SHERATON

Elegant by Design: Javascript in Javascript

PAOLO CARESSA

IT Expert - QA Manager @ GSE spa

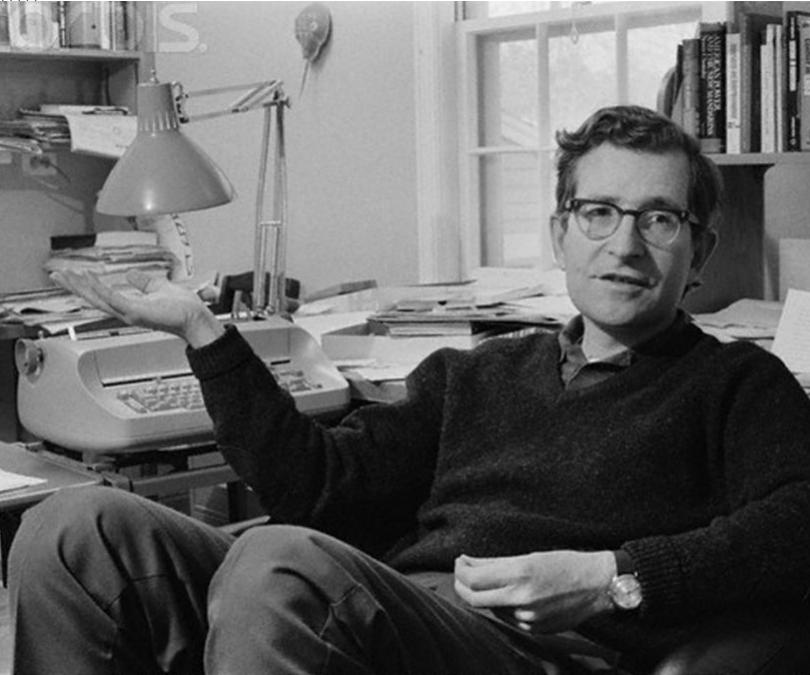
() CODERFUL



LET'S
TALK
CODE

Linguaggi, ricorsione, autoreferenzialità

Il fascino (discreto) dei linguaggi di programmazione

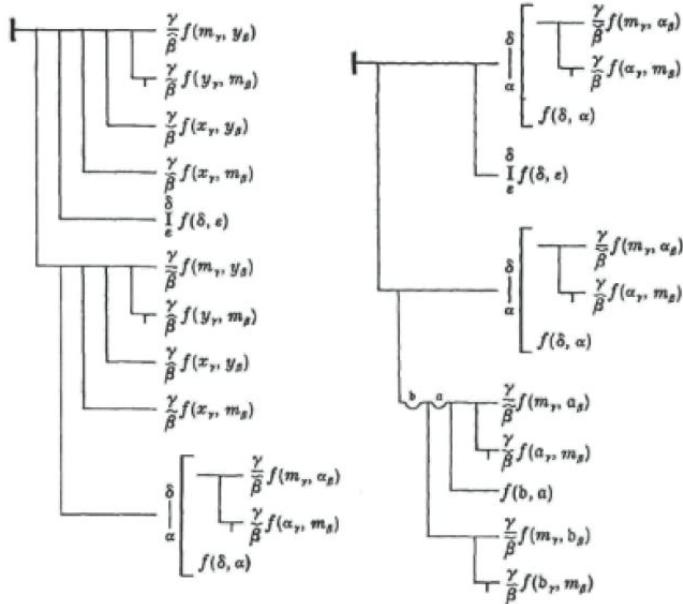


I linguaggi di programmazione sono esempi di sistemi formali.

Possiamo *generare* la totalità delle stringhe che formano programmi corretti.

Linguaggi formali vs. automi

A cominciare sono stati i logici



“Formalismo da alas ad mente de homo”

- 0 $N_0 \varepsilon \text{ Cls}$
- 1 $0 \varepsilon N_0$
- 2 $a \varepsilon N_0 \supset a + \varepsilon N_0$
- 3 $s \varepsilon \text{ Cls} . 0 \varepsilon s : a \varepsilon s \supset a . a + \varepsilon s : \supset N_0 \supset s$
- 4 $a, b \varepsilon N_0 . a + = b + \supset a = b$
- 5 $a \varepsilon N_0 \supset a + == 0$



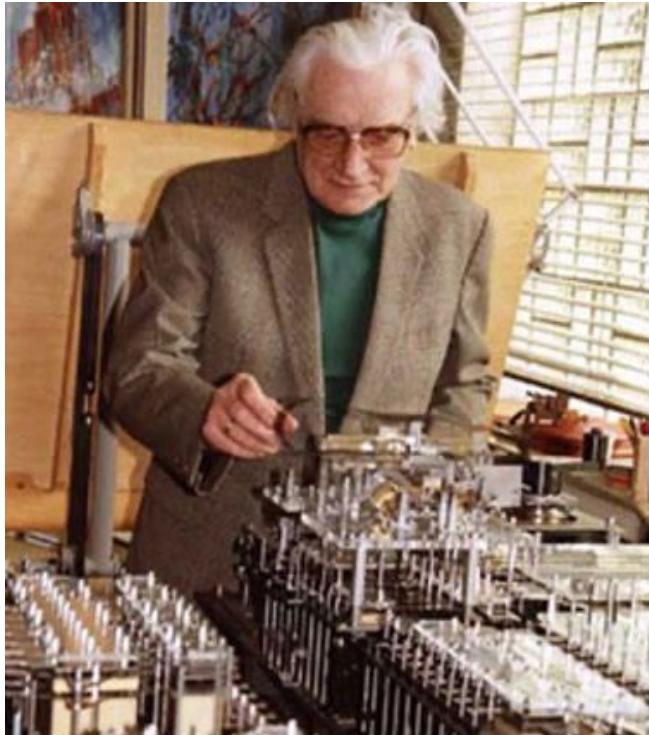
Il linguaggio formale per eccellenza



Hilbert

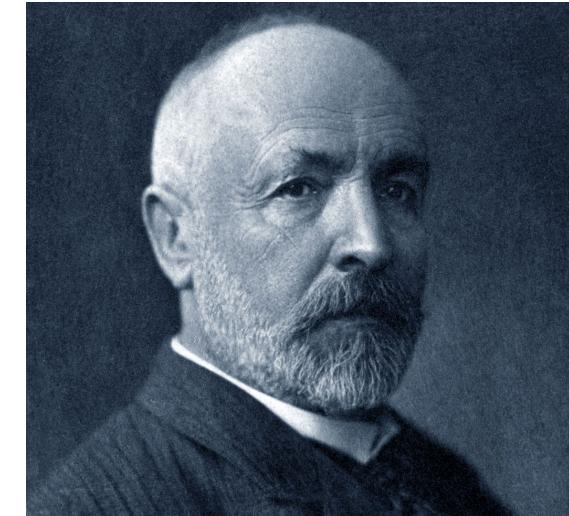
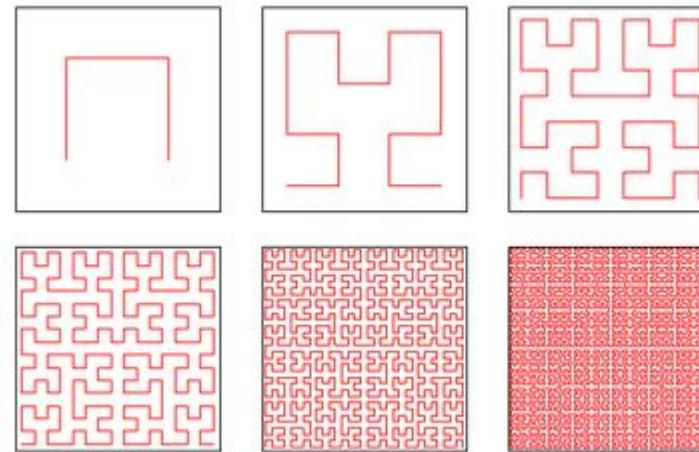
	$\frac{B \vdash B \quad C \vdash C}{B \vee C \vdash B, C} (I)$
	$\frac{B \vee C \vdash B, C}{B \vee C \vdash C, B} (\vee L)$
	$\frac{B \vee C \vdash C, B}{B \vee C, \neg C \vdash B} (\neg L)$
	$\frac{B \vee C, \neg C \vdash B}{(B \vee C), \neg C, (B \rightarrow \neg A) \vdash \neg A} (\rightarrow L)$
	$\frac{(B \vee C), \neg C, (B \rightarrow \neg A) \vdash \neg A}{(B \vee C), \neg C, ((B \rightarrow \neg A) \wedge \neg C) \vdash \neg A} (\wedge L_1)$
	$\frac{(B \vee C), \neg C, ((B \rightarrow \neg A) \wedge \neg C) \vdash \neg A}{(B \vee C), ((B \rightarrow \neg A) \wedge \neg C), \neg C \vdash \neg A} (\wedge L_2)$
$A \vdash A$	$\frac{}{(B \vee C), ((B \rightarrow \neg A) \wedge \neg C), ((B \rightarrow \neg A) \wedge \neg C) \vdash \neg A} (\neg R)$
$\vdash \neg A, A$	$\frac{}{(B \vee C), ((B \rightarrow \neg A) \wedge \neg C) \vdash \neg A} (CL)$
$\vdash A, \neg A$	$\frac{}{((B \rightarrow \neg A) \wedge \neg C), (B \vee C) \vdash \neg A} (PL)$
	$\frac{((B \rightarrow \neg A) \wedge \neg C), (A \rightarrow (B \vee C)) \vdash \neg A, \neg A}{((B \rightarrow \neg A) \wedge \neg C), (A \rightarrow (B \vee C)) \vdash \neg A, \neg A} (\rightarrow L)$

Il primo linguaggio di programmazione



V	$\begin{array}{c} V \Rightarrow Z \\ 0 \end{array}$	
S	$m \times (\sigma, \tau)$	$m \times (\sigma, \tau)$
V	$W1(m-1) \left[\begin{array}{c} Z \Rightarrow Z \\ 0 \end{array} \right]$	$i \Rightarrow \varepsilon$
K	$i+1$	
S	$(\sigma, \tau) \quad (\sigma, \tau)$	$1.n \quad 1.n$
V	$W \left[\varepsilon \geq 0 \rightarrow \begin{array}{c} Z < Z \vee (Z = Z \wedge Z < Z) \Rightarrow Z \\ 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 2 \\ 0 \quad \varepsilon.0 \quad 0 \quad \varepsilon.0 \quad 1 \quad \varepsilon.1 \end{array} \right]$	
K	$\sigma \quad \sigma \quad \sigma \quad \sigma$	
S	$Z \rightarrow \left[\begin{array}{c} Z \Rightarrow Z \\ 0 \quad 0 \\ \varepsilon \quad \varepsilon + 1 \end{array} \right]$	$\varepsilon - 1 \Rightarrow \varepsilon$
V	$(\sigma, \tau) \quad (\sigma, \tau)$	
K	$\overline{Z} \rightarrow \left[\begin{array}{c} Z \Rightarrow Z \\ 1 \quad 0 \\ \varepsilon \quad \varepsilon + 1 \end{array} \right]$	Fin^3
S	$\sigma \quad \sigma$	
V	$\varepsilon = -1 \rightarrow Z \Rightarrow Z$	
K	$1 \quad 0$	
S	0	
V	$Z \Rightarrow R$	
S	0	
V	$m \times \sigma$	$m \times \sigma$
S		

To iterate is human, to recurse is divine



La sintassi dei linguaggi è ricorsiva

<espressione> = <operando>

| <operando> <operatore> <espressione>

<operando> = <numero> | <variabile>

| (<espressione>)

<numero> = <cifra> | <numero> <cifra>

<operatore> = + | - | * | /



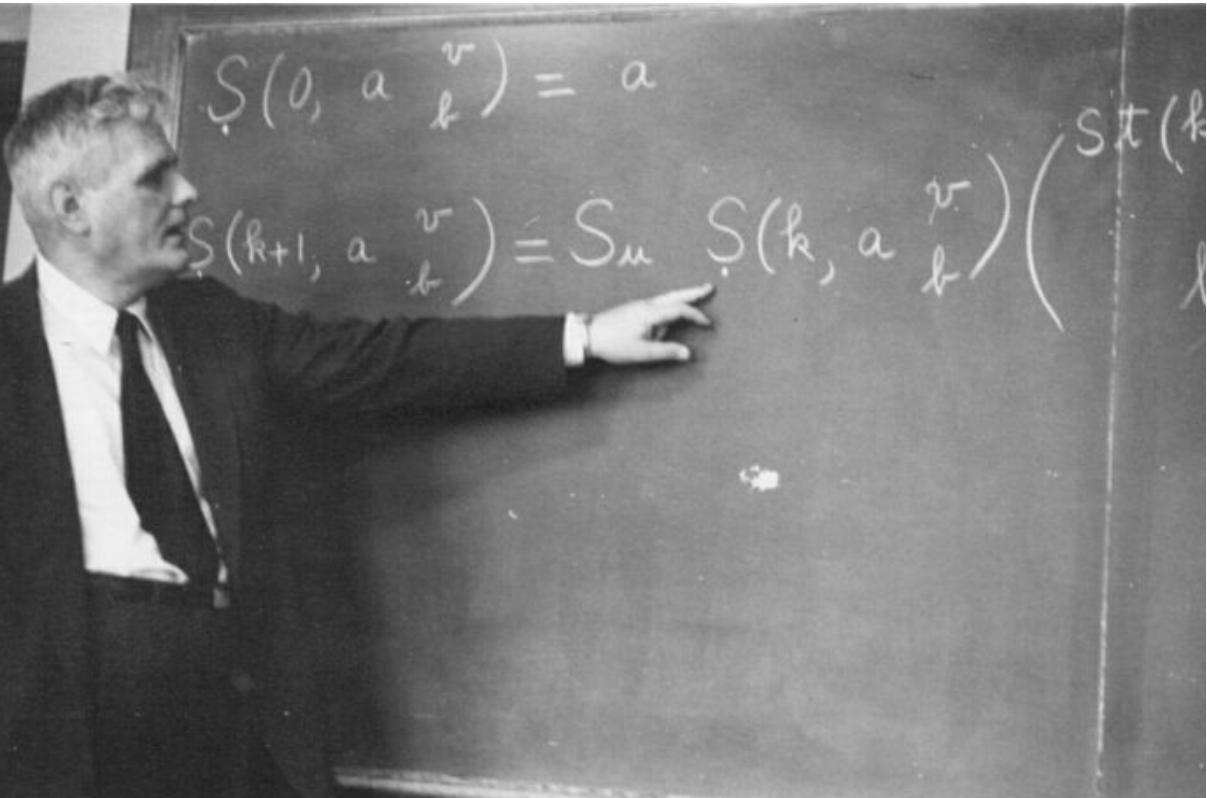
Linguaggi che esprimono sé stessi

Teorema di incompletezza di Gödel. Supponendo che \mathbf{PA}^1 sia coerente e che D^* sia rappresentabile allora esiste un enunciato che non sta né in D né in R .

Dimostrazione. Supponiamo che la formula \mathbf{F} rappresenti D^* e consideriamo $\neg\mathbf{F}$: questa avrà un numero di Gödel $b = g_{\mathbf{F}}$. Dato che \mathbf{F} rappresenta D^* , $\mathbf{F}[\bar{b}]$ è dimostrabile (sta in D) se e solo se $b \in D^*$ che equivale a dire che $\mathbf{F}_b[\bar{b}] \in D$, cioè che $\mathbf{F}_b[\bar{b}]$ è dimostrabile. Ma $\mathbf{F}_b[\bar{b}]$ è la formula $\neg\mathbf{F}[\bar{b}]$! Quindi $\mathbf{F}[\bar{b}]$ è dimostrabile se e solo se lo è $\neg\mathbf{F}[\bar{b}]$; attenzione: non abbiamo dimostrato che $\mathbf{F}[\bar{b}] \in D$ ma che $\mathbf{F}[\bar{b}] \in D$ se e solo se $\neg\mathbf{F}[\bar{b}] \in D$, pertanto delle due l'una, o $\mathbf{F}[\bar{b}]$ e $\neg\mathbf{F}[\bar{b}]$ sono entrambe dimostrabili, o non lo è nessuna delle due. L'ipotesi di coerenza di \mathbf{PA}^1 esclude la prima conclusione e quindi siamo costretti ad ammettere che l'enunciato $\mathbf{F}[\bar{b}]$ non è né dimostrabile né refutabile. \square

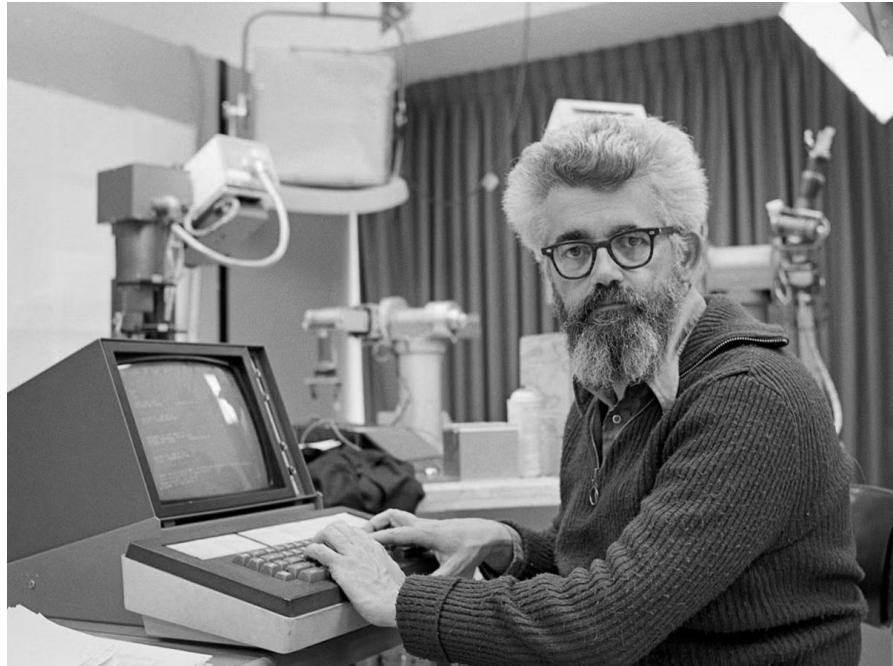


Algoritmi, macchine e linguaggi

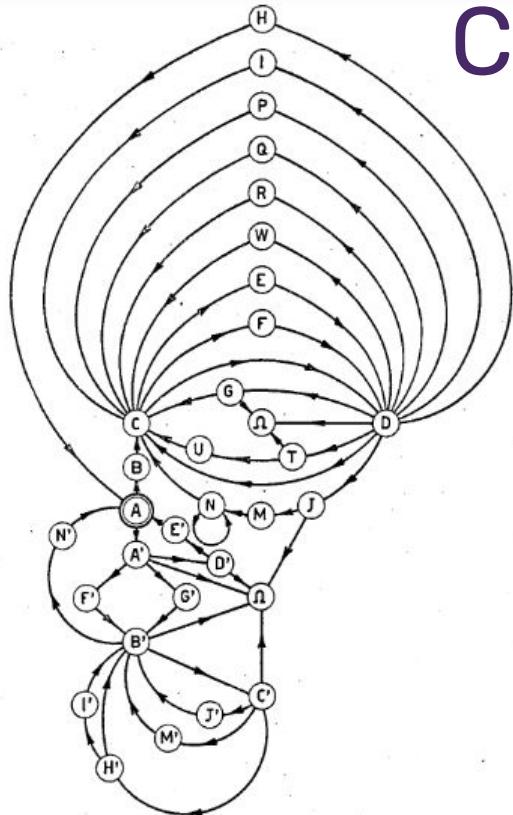


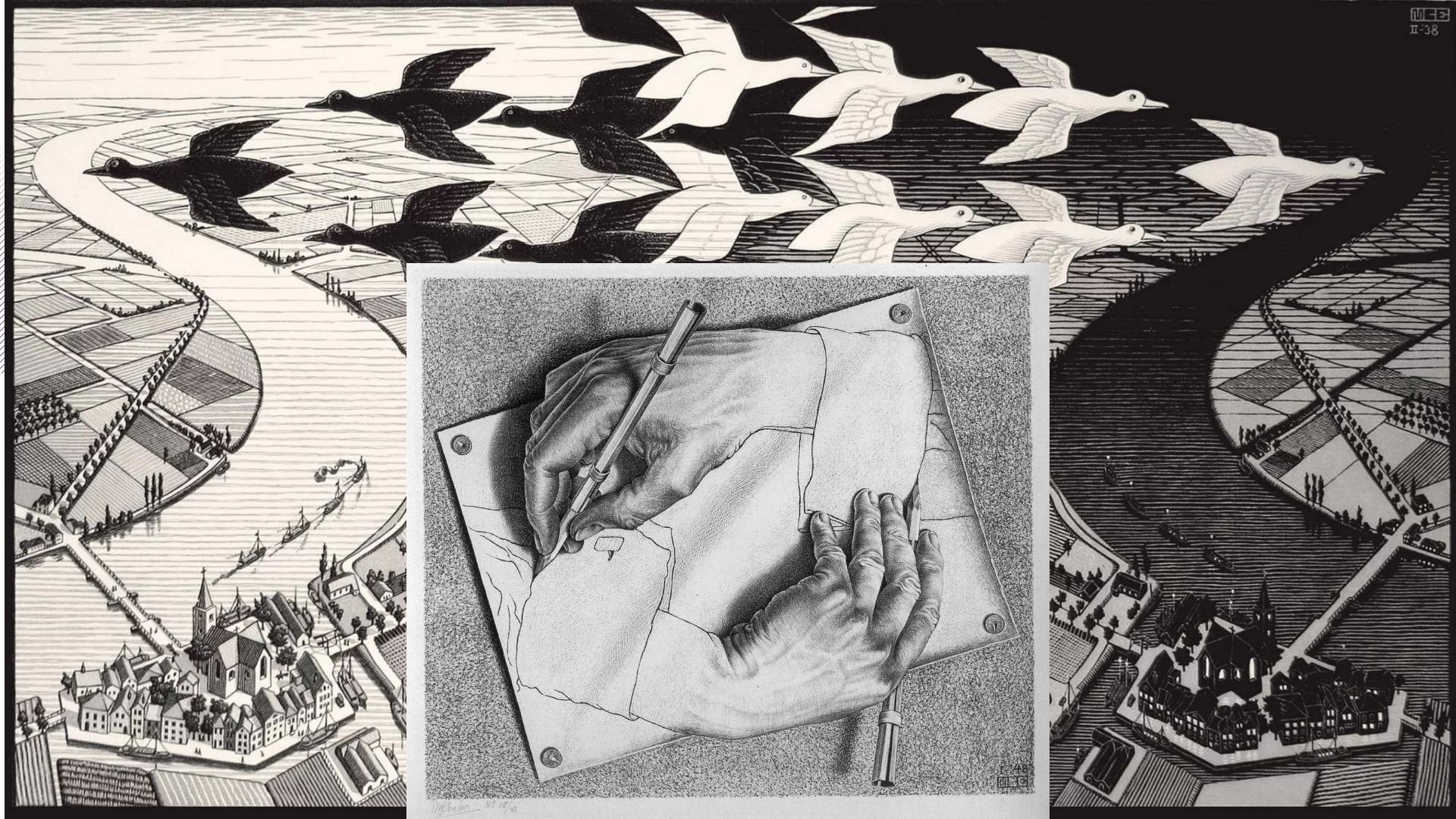
```
(LABEL EVAL (LAMBDA (E A)
  (COND ((ATOM E)
    (COND ((EQ E NIL) NIL)
      ((EQ E T) T)
      (T (CDR ((LABEL
        ASSOC
        (LAMBDA (E A)
          (COND ((NULL A) NIL)
            ((EQ E (CAAR A)) (CAR A))
            (T (ASSOC E (CDR A)))))))
        E
      R))))))
  ((ATOM (CAR E))
    (COND ((EQ (CAR E) (QUOTE QUOTE)) (CDDR E))
      ((EQ (CAR E) (QUOTE CAR))
        (CAR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CDR))
        (CDR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CADR))
        (CADR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CADDR))
        (CADDR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CAAR))
        (CAAR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CADDR))
        (CADDR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CADAR))
        (CADAR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CADDAR))
        (CADDAR (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE ATOM))
        (ATOM (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE NULL))
        (NULL (EVAL (CDDR E) A)))
      ((EQ (CAR E) (QUOTE CONS))
        (CONS (EVAL (CDDR E) A) (EVAL (CADDR E) A)))
      ((EQ (CAR E) (QUOTE EQ))
        (EQ (EVAL (CADDR E) A) (EVAL (CADDR E) A)))
      ((EQ (CAR E) (QUOTE COND))
        ((LABEL EVCOND
          (LAMBDA (U A) (COND ((EVRL (CAAR U) A)
            (EVRL (CADAR U)
              A))
            (T (EVCOND (CDR U)
```

Lisp in Lisp (1958)



Il compilatore meta-circolare di Corrado Böhm (1951)





Un linguaggio è elegante se
può implementare sé stesso
in modo naturale

_FRONTEND EDITION

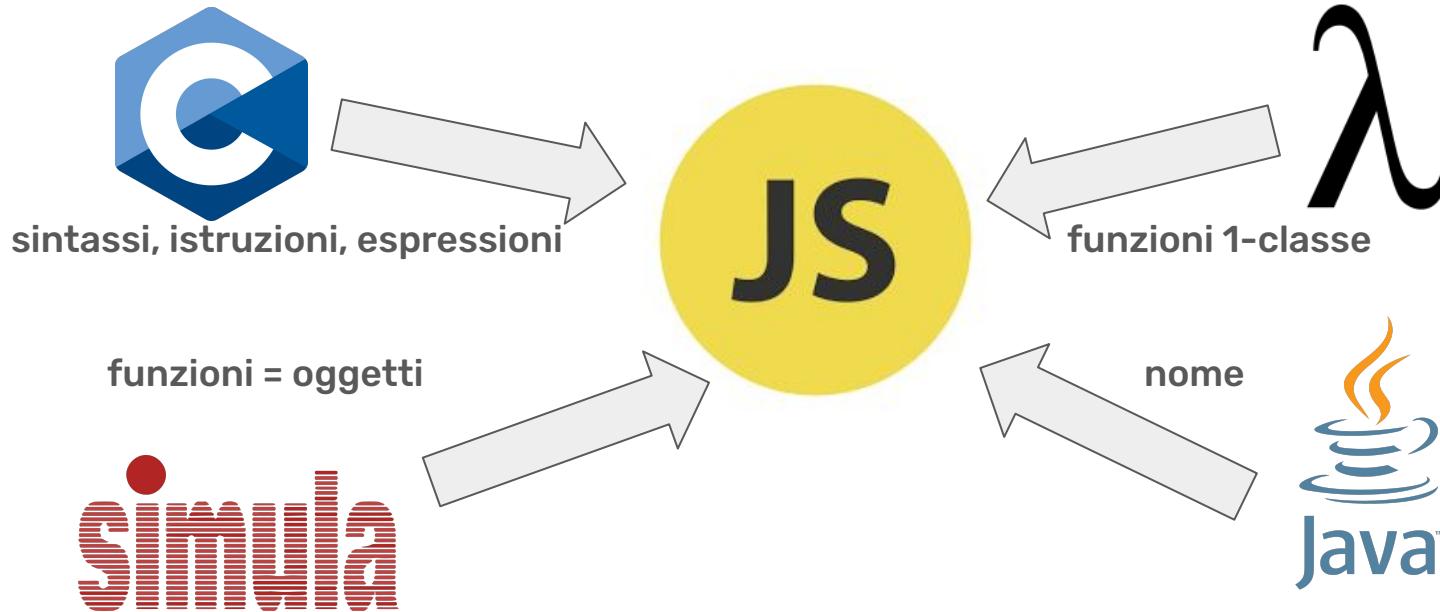
// CATANIA

28.06.2024

FOUR POINTS
BY SHERATON

Javascript

Un incontro felice



Di che Javascript stiamo parlando?

```
program      = {statement}
statement    = expression ";" 
              | "do" "{" program "}" "while" "(" expression ")" ";" 
              | "for" "(" [initialize] ";" [expression] ";" [initialize] ")" "{" program "}" 
              | "for" "(" "let" name "in" expression ")" "{" program "}" 
              | "if" "(" expression ")" "{" program "}" ["else" "{" program "}" ]
              | "let" name ["/=" expression] {," name ["/=" expression]} ";" 
              | "return" [expression] ";" 
              | "throw" expression ";" 
              | "while" "(" expression ")" "{" program "}" 
initialize   = "let" name "=" expression | expression
expression   = {prefix_op} value [binary_op expression]
value        = constant
              | object
              | variable
              | "(" expression ")"
```

Di che Javascript stiamo parlando?

```
constant      = digit{digit}["."{digit}]
              | "null"
              | "undefined"
              | "true"
              | "false"
              | '''{character}'''
              | '''{character}'''

object        = "{" name ":" expression "," name ":" expression } "
              | "[" [expression "," expression] "]"
              | "function" "(" [name "," name] ")" "{" program "}"
              | "new" value

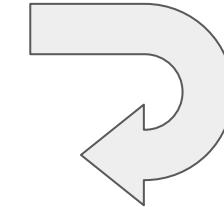
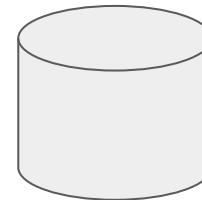
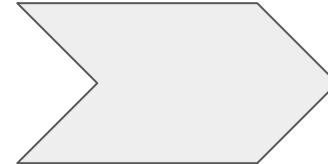
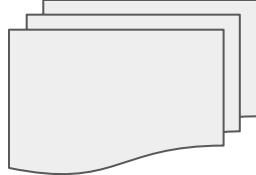
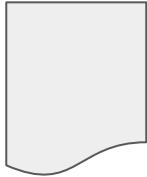
variable       = name
              | variable "." name
              | variable "[" expression "]"
              | variable "(" [expression "," expression] ")"

prefix_op     = "-" | "++" | "--" | "!"

binary_op     = "+" | "-" | "*" | "/" | "==" | "<" | ">" | "!=" | ">=" | "<="
              | "&&" | "||" | "=" | "+=" | "-="
```

Struttura del compilatore

Come si scrive un compilatore?



TEXT

SCANNER

TOKENS

COMPILER

CODE

CPU/VM

In Javascript...

```
run(compile(scan("alert('Hello World')")))
```

Lo scanner

function scan(text): accetta un testo e restituisce una token list, cioè un array di oggetti

```
token = { s: "rappresentazione stringa",  
          t: "tipo (number, string, ...)",  
          l: numero di riga nel testo,  
          c: numero di colonna nel testo }
```

Mostra esempio live!

Il compilatore vero e proprio

function compile(tl): accetta una token list e restituisce un oggetto di tipo "runtime"

```
rt = { stack: [...],  
       env: {v1:a1, ...},  
       code: [i1, ...], ic: indice in c,  
       dump: [...] }
```

Mostra esempio live!

Ispirato alla macchina virtuale SECD



Proposta nel 1963 da Peter J. Landin per descrivere l'interpretazione di espressioni in un linguaggio funzionale puro.

[Landmark Landin paper](#)

In Javascript è facilissima.

La macchina virtuale

function run(rt): accetta un oggetto di tipo "runtime" e lo esegue.

Questa è facile da scrivere una volta definito il set di istruzioni

Mostra esempio live!

compile e run vanno a braccetto

Il compilatore deve conoscere non solo il linguaggio sorgente ma anche il linguaggio target: per questo solitamente si separa in due componenti, il front-end e il back-end del compilatore. Più back-end compilano per macchine diverse uno stesso linguaggio di front-end.

Nel nostro caso il compilatore è monolitico.

Mostra il codice!!!

Compilare le istruzioni

Metodo top-down

Una istruzione inizia con una keyword e termina con un ";". Se non è così la consideriamo una espressione.

```
if (token == "do") {compila un ciclo do-while;}  
elif (token == "if") {compila un if-else;}  
elif (token == "let") {compila una assegnazione;}  
...  
else {compila una espressione;}
```

Mostra il codice!!!

Metodo top-down

Ogni istruzione ramifica in sotto-compilazioni, es:

compila ciclo do-while:

a = rt.code

compila un blocco;

deve esserci "while" (se no errore);

compila espressione;

compila "JPZ a".

Istruzioni di controllo => salti!

Ogni istruzione di controllo si esprime in termini di combinazioni salti incondizionati (JP) e condizionati (JP, JPNZ) .

Mostralo sul compilatore!!!

Compilare le espressioni

Metodo bottom-up

1 + x * (x - 1) => c = [] s = []

Metodo bottom-up

```
1 + x * (x - 1) => c = []           s = []
+ x * (x - 1)    => c = [1]         s = []
```

Metodo bottom-up

```
1 + x * (x - 1) => c = []           s = []
+ x * (x - 1)    => c = [1]         s = []
x * (x - 1)     => c = [1]         s = [+]
```

Metodo bottom-up

```
1 + x * (x - 1) => c = []           s = []
+ x * (x - 1)    => c = [1]         s = []
x * (x - 1)     => c = [1]         s = [+]
* (x - 1)        => c = [1 x]       s = [+]
```

Metodo bottom-up

1 + x * (x - 1) => c = [] s = []
+ x * (x - 1) => c = [1] s = []
x * (x - 1) => c = [1] s = [+]
* (x - 1) => c = [1 x] s = [+]
(x - 1) => c = [1 x] s = [+ *]

Metodo bottom-up

1 + x * (x - 1) => c = [] s = []
+ x * (x - 1) => c = [1] s = []
x * (x - 1) => c = [1] s = [+]
* (x - 1) => c = [1 x] s = [+]
(x - 1) => c = [1 x] s = [+ *]
x - 1) => c = [1 x] s = [+ * ()]

Metodo bottom-up

1 + x * (x - 1) => c = [] s = []
+ x * (x - 1) => c = [1] s = []
x * (x - 1) => c = [1] s = [+]
* (x - 1) => c = [1 x] s = [+]
(x - 1) => c = [1 x] s = [+ *]
x - 1) => c = [1 x] s = [+ * ()]
- 1) => c = [1 x x] s = [+ * ()]

Metodo bottom-up

1 + x * (x - 1) => c = [] s = []
+ x * (x - 1) => c = [1] s = []
x * (x - 1) => c = [1] s = [+]
* (x - 1) => c = [1 x] s = [+]
(x - 1) => c = [1 x] s = [+ *]
x - 1) => c = [1 x] s = [+ * ()]
- 1) => c = [1 x x] s = [+ * ()]
1) => c = [1 x x 1] s = [+ * (-)]

Metodo bottom-up

1 + x * (x - 1) => c = [] s = []
+ x * (x - 1) => c = [1] s = []
x * (x - 1) => c = [1] s = [+]
* (x - 1) => c = [1 x] s = [+]
(x - 1) => c = [1 x] s = [+ *]
x - 1) => c = [1 x] s = [+ * ()]
- 1) => c = [1 x x] s = [+ * ()]
1) => c = [1 x x 1] s = [+ * (-)]
) => c = [1 x x 1] s = [+ * (-)]

Metodo bottom-up

1 + x * (x - 1) => c = [] s = []
+ x * (x - 1) => c = [1] s = []
x * (x - 1) => c = [1] s = [+]
* (x - 1) => c = [1 x] s = [+]
(x - 1) => c = [1 x] s = [+ *]
x - 1) => c = [1 x] s = [+ * ()]
- 1) => c = [1 x x] s = [+ * ()]
1) => c = [1 x x 1] s = [+ * (-)]
) => c = [1 x x 1] s = [+ * (-)]

=> c = [1 x x 1 - * +]

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

`c = [1 x x 1 - * +] => s = [1]`

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

`c = [1 x x 1 - * +] => s = [1]`

`c = [1 x x 1 - * +] => s = [1 2]`

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

`c = [1 x x 1 - * +] => s = [1]`

`c = [1 x x 1 - * +] => s = [1 2]`

`c = [1 x x 1 - * +] => s = [1 2 2]`

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

`c = [1 x x 1 - * +] => s = [1]`

`c = [1 x x 1 - * +] => s = [1 2]`

`c = [1 x x 1 - * +] => s = [1 2 2]`

`c = [1 x x 1 - * +] => s = [1 2 2 1]`

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

`c = [1 x x 1 - * +] => s = [1]`

`c = [1 x x 1 - * +] => s = [1 2]`

`c = [1 x x 1 - * +] => s = [1 2 2]`

`c = [1 x x 1 - * +] => s = [1 2 2 1]`

`c = [1 x x 1 - * +] => s = [1 2 1]`

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

```
c = [1 x x 1 - * +] => s = [1]
c = [1 x x 1 - * +] => s = [1 2]
c = [1 x x 1 - * +] => s = [1 2 2]
c = [1 x x 1 - * +] => s = [1 2 2 1]
c = [1 x x 1 - * +] => s = [1 2 1]
c = [1 x x 1 - * +] => s = [1 2]
```

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

```
c = [1 x x 1 - * +] => s = [1]
c = [1 x x 1 - * +] => s = [1 2]
c = [1 x x 1 - * +] => s = [1 2 2]
c = [1 x x 1 - * +] => s = [1 2 2 1]
c = [1 x x 1 - * +] => s = [1 2 1]
c = [1 x x 1 - * +] => s = [1 2]
c = [1 x x 1 - * +] => s = [3]
```

Valutazione in forma postfissa

Supponiamo `env = {x:2}`

`c = [1 x x 1 - * +] => s = [1]`

`c = [1 x x 1 - * +] => s = [1 2]`

`c = [1 x x 1 - * +] => s = [1 2 2]`

`c = [1 x x 1 - * +] => s = [1 2 2 1]`

`c = [1 x x 1 - * +] => s = [1 2 1]`

`c = [1 x x 1 - * +] => s = [1 2]`

`c = [1 x x 1 - * +] => s = [3]`

Cioè il risultato di `((x)=>(1+x*(x-1))) (2)`

Paolo Caressa @ [GSE spa](#)

<https://github.com/pcaressa/>

<https://linkedin.com/in/paolocarella>

https://twitter.com/www_caressa_it

Grazie

APPLAUSI LIBERI



_FRONTEND EDITION

// CATANIA

28.06.2024

FOUR POINTS
BY SHERATON

({}) CODERFUL