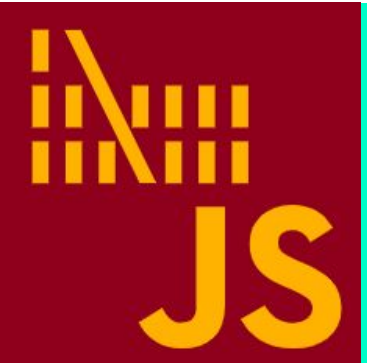
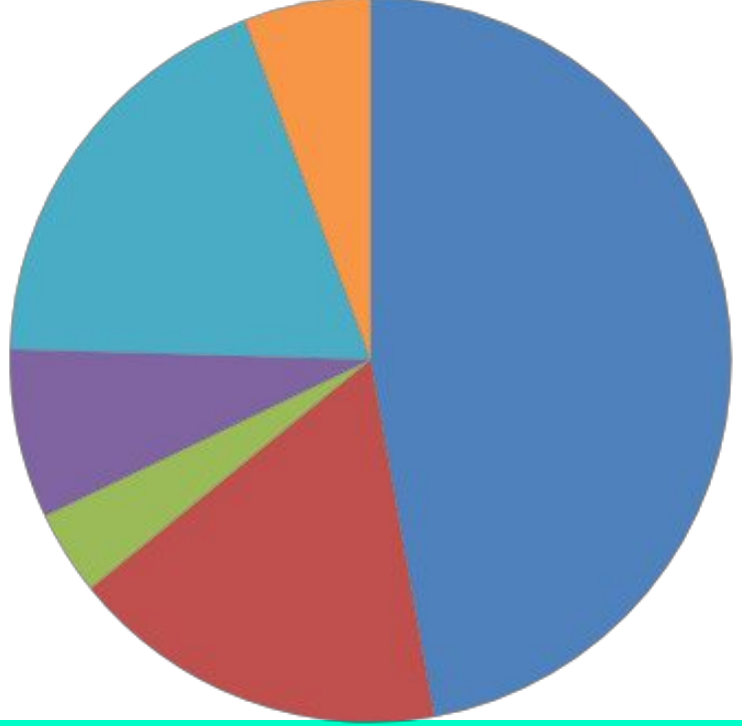
Elegant by design: Javascript in Javascript



Paolo Caressa PhD - 18 settembre 2024

(IT Expert|QA Manager) @ GSE spa



Una vita con… torta

Paolo Caressa

Linguaggi, ricorsione, autoreferenzialità

Il fascino (discreto) dei linguaggi di programmazione

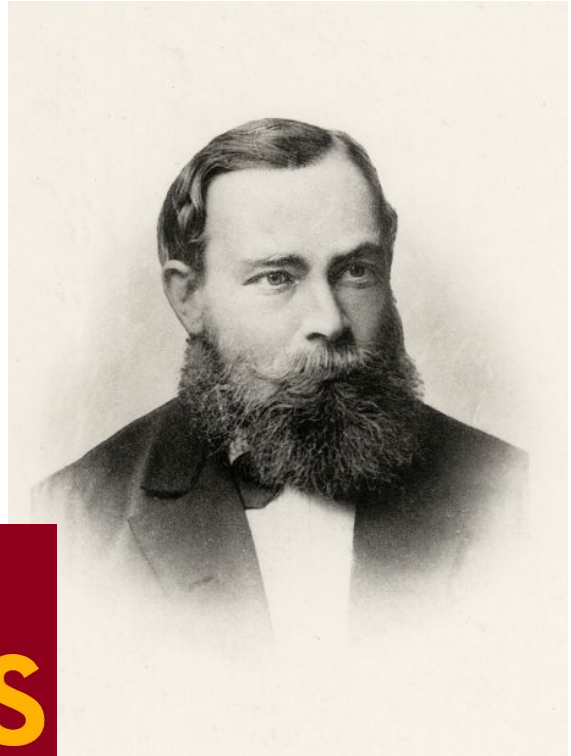
Noam Chomsky

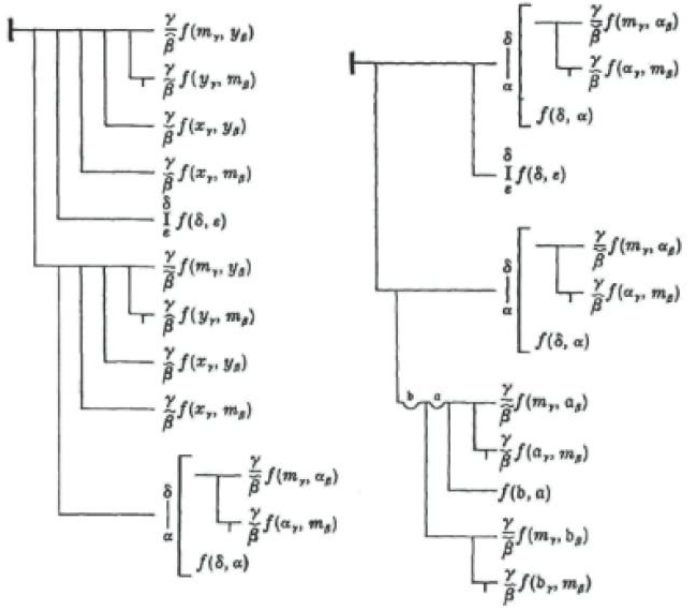
I linguaggi di programmazione sono esempi di sistemi formali.

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

Linguaggi formali vs. automi

A cominciare sono stati i logici





Gottlob Frege *Begriffsschrift*

“Formalismo da alas ad mente de homo”

·0 $N_0 \ \varepsilon \ \mathrm{Cls}$

·1 $0 \ \varepsilon \ N_0$

·2 $a\varepsilon \ N_0 \ . \supset . \ a+ \ \varepsilon \ N_0$

·3 $s\varepsilon \ \mathrm{Cls} \ . \ 0\varepsilon s : a\varepsilon s \ . \supset a \ . \ a+ \ \varepsilon s : \supset . \ N_0 \supset s$

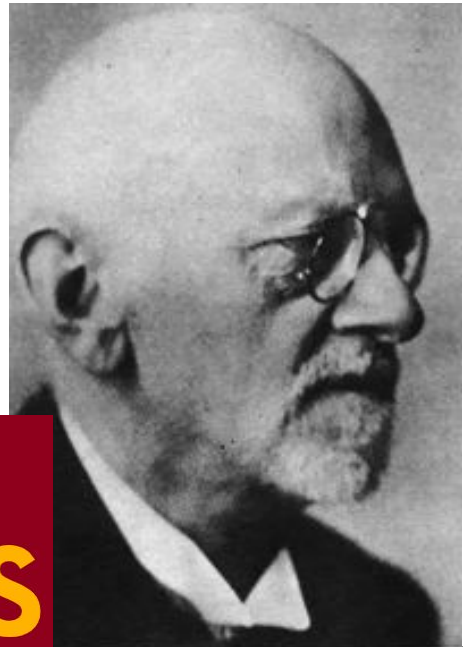
·4 $a,b\varepsilon \ N_0 \ . \ a+ = b+ \ . \supset . \ a = b$

·5 $a\varepsilon \ N_0 \ . \supset . \ a+ \neq 0$

Giuseppe Peano *Formulaire de mathématiques*

Il linguaggio formale per eccellenza

David Hilbert *Calcolo dei predicati del primo ordine*



$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}(I) \quad \dfrac{}{C \vdash C}(I)}{B \vee C \vdash B, C}(\vee L)}{B \vee C \vdash C, B}(PR)}{B \vee C, \neg C \vdash B}(\neg L) \quad \dfrac{}{\neg A \vdash \neg A}(I)}{(B \vee C), \neg C, (B \to \neg A) \vdash \neg A}(\to L)}{(B \vee C), \neg C, ((B \to \neg A) \wedge \neg C) \vdash \neg A}(\wedge L_1)}{(B \vee C), ((B \to \neg A) \wedge \neg C), \neg C \vdash \neg A}(PL)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}(I)}{\vdash \neg A, A}(\neg R)}{\vdash A, \neg A}(PR) \quad \dfrac{(B \vee C), ((B \to \neg A) \wedge \neg C), ((B \to \neg A) \wedge \neg C) \vdash \neg A}{(B \vee C), ((B \to \neg A) \wedge \neg C) \vdash \neg A}(CL)}{((B \to \neg A) \wedge \neg C), (B \vee C) \vdash \neg A}(PL)}{((B \to \neg A) \wedge \neg C), (A \to (B \vee C)) \vdash \neg A, \neg A}(\to L)^{7}$$

Il primo(?) linguaggio di programmazione

Konrad Zuse *Plankalkül*

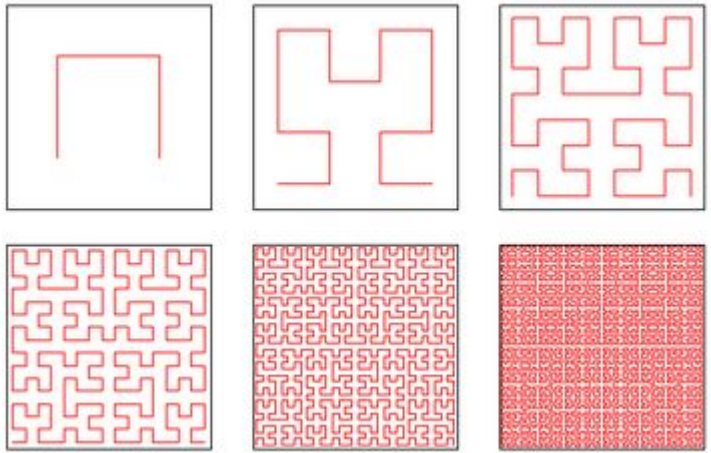
$$
\begin{array}{l|l}
 & V \quad \Rightarrow Z \\
V & 0 \quad 0 \\
S & m \times (\sigma,\tau) \quad m \times (\sigma,\tau)
\end{array}
$$

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

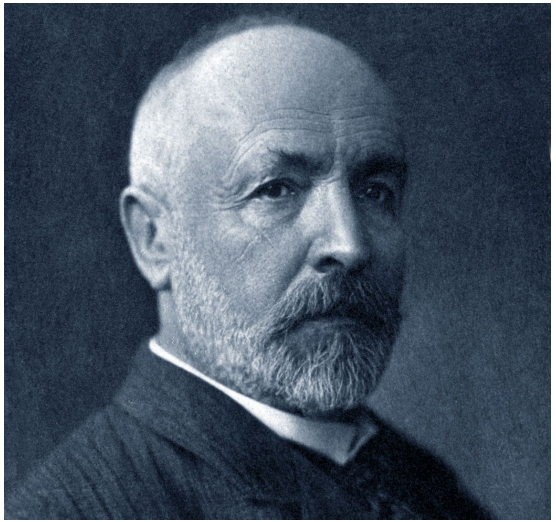
$$
\begin{array}{l|l}
 & Z \quad \Rightarrow R \\
V & 0 \quad 0 \\
S & m \times \sigma \quad m \times \sigma
\end{array}
$$

To iterate is human, to recurse is divine

Richard Dedekind



Georg Cantor



La sintassi dei linguaggi è ricorsiva

<espressione> = <operando>
| <operando> <operatore> <espressione>

<operando> = <numero> | <variabile>
| (<espressione>)

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

<operatore> = + | - | * | /

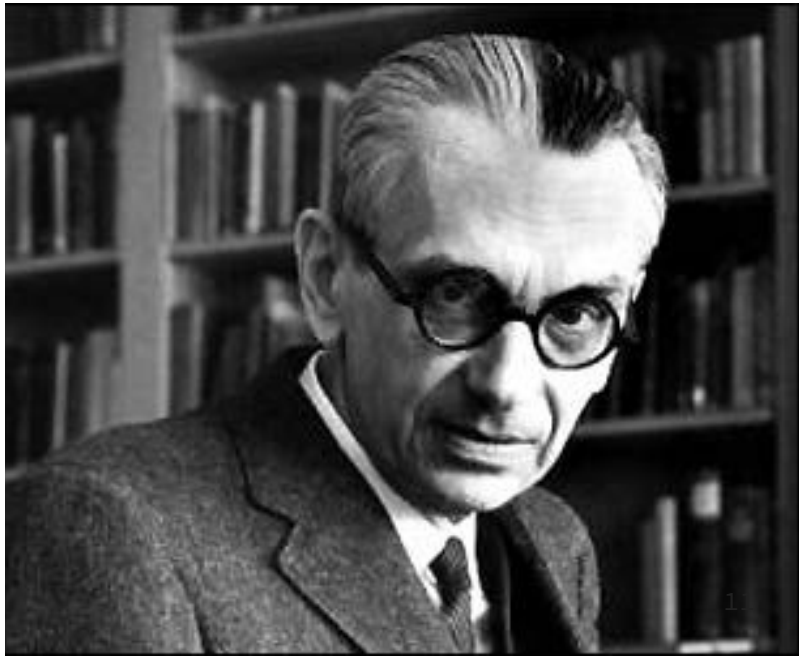
John McCarthy, Friedrich Bauer, Joseph Wegstein.
John Backus, Peter Naur, Alan Perlis.

Linguaggi che rappresentano se 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.

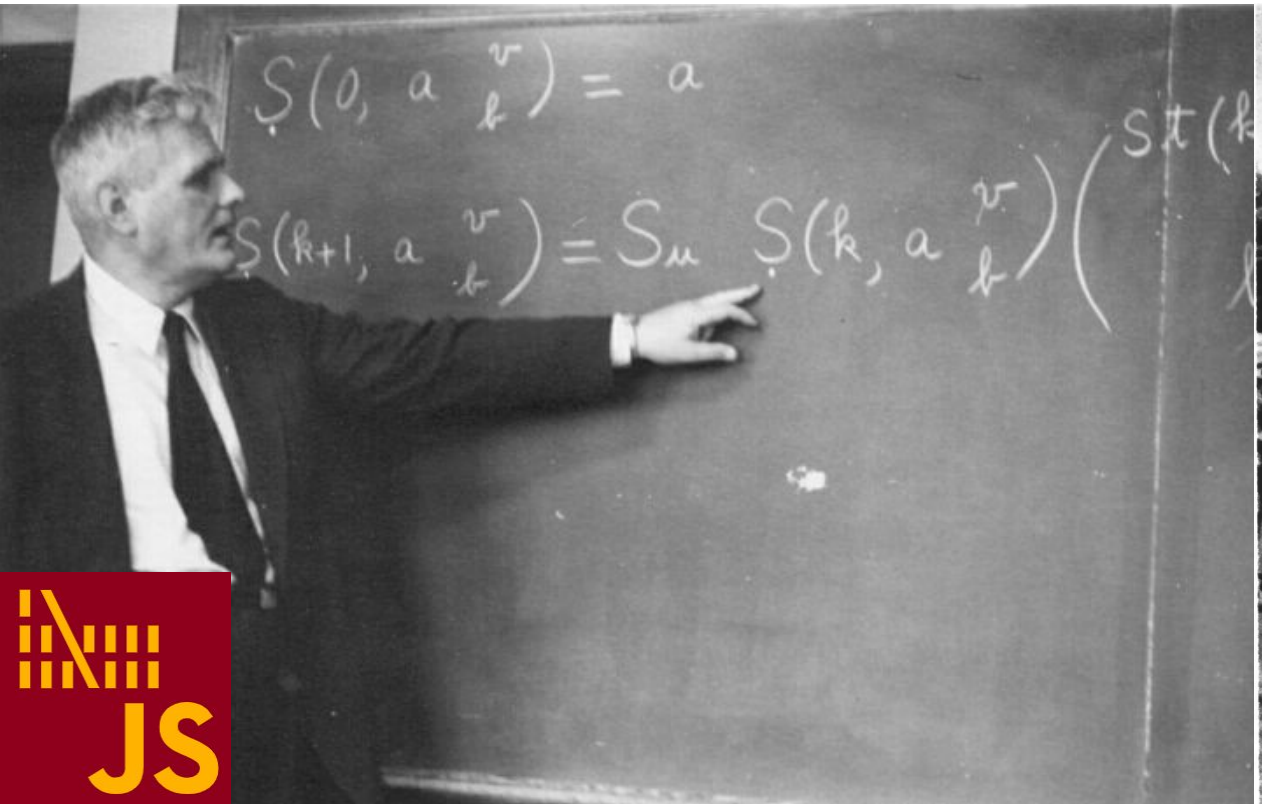
Dimostrazione. Supponiamo che la formula $\mathbf{F}$ rappresenti D^* e consideriamo $\neg\mathbf{F}$: questa avrà un numero di Gödel $b = g_{\mathbf{F}}$. Dato che $\mathbf{F}$ rappresenta D^*, $\mathbf{F}[\bar{b}]$ è dimostrabile (sta in D) se e solo se $b \in D^*$ che equivale a dire che $\mathbf{F_b}[\bar{b}] \in D$, 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. □

Kurt Gödel



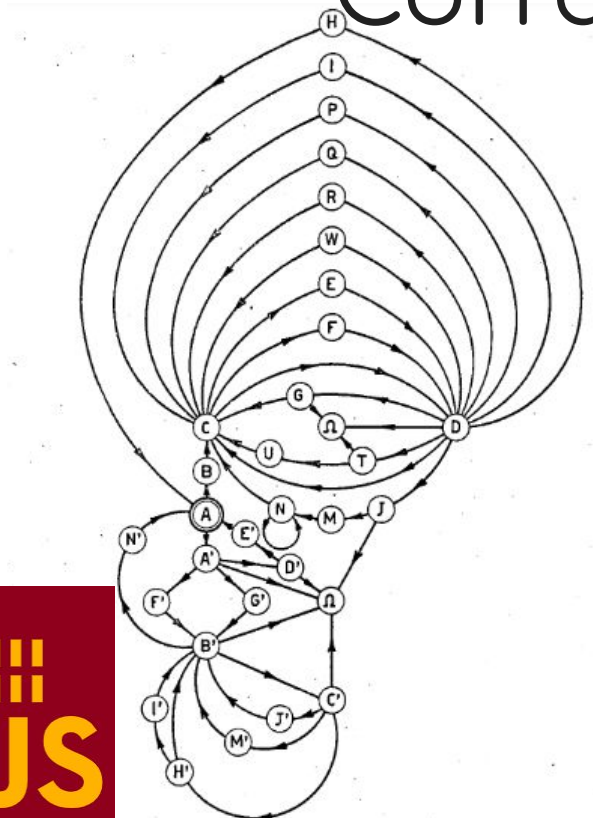
Algoritmi, automi e linguaggi

Alonzo Church



Alan Turing

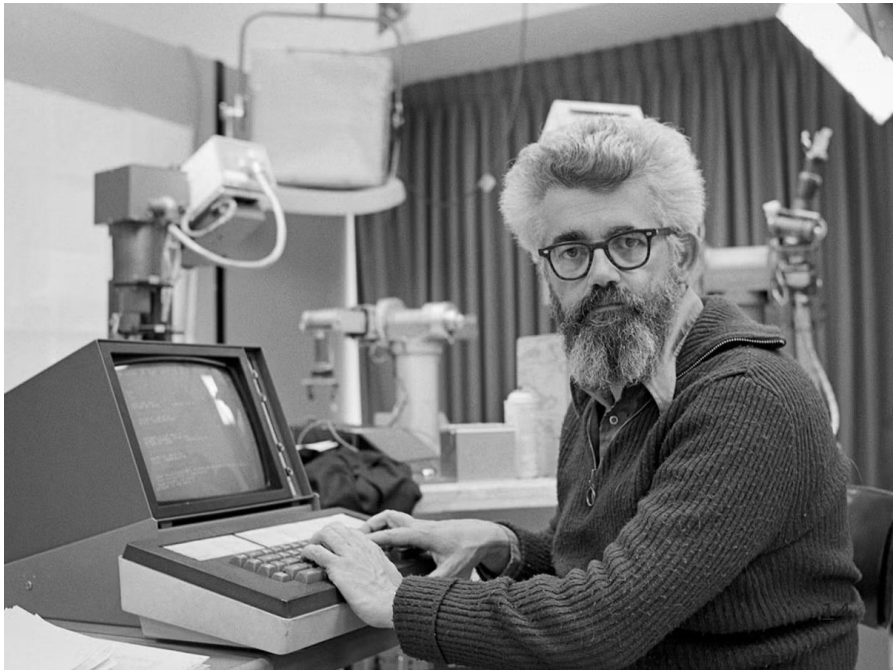
Il compilatore meta-circolare di Corrado Böhm (1954)



Lisp in Lisp (1958)

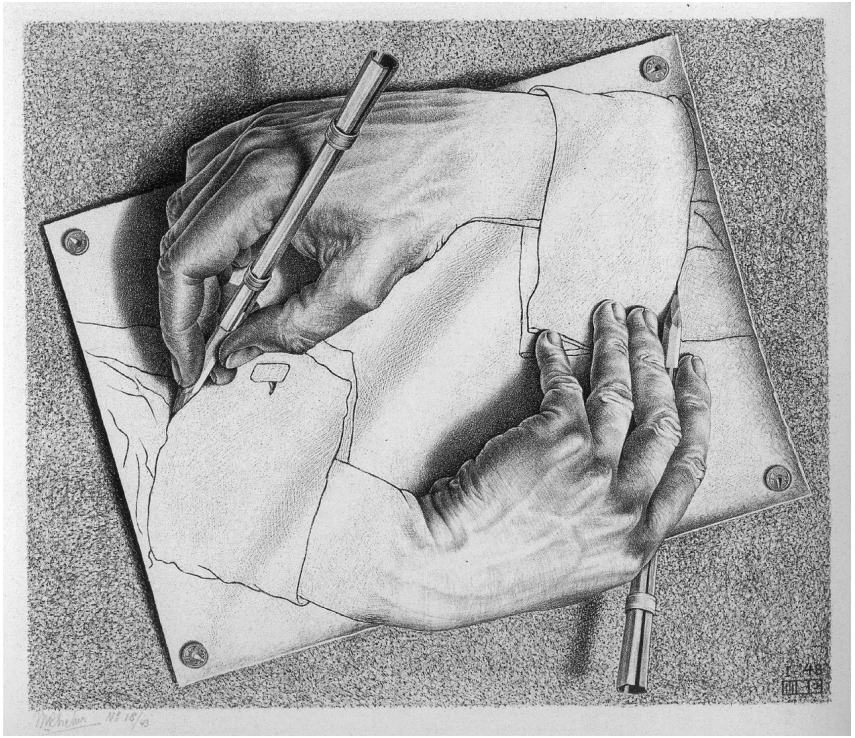
John McCarthy

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

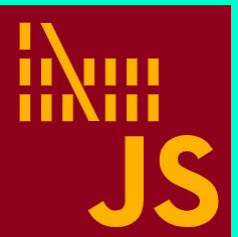


Principio guida estetico

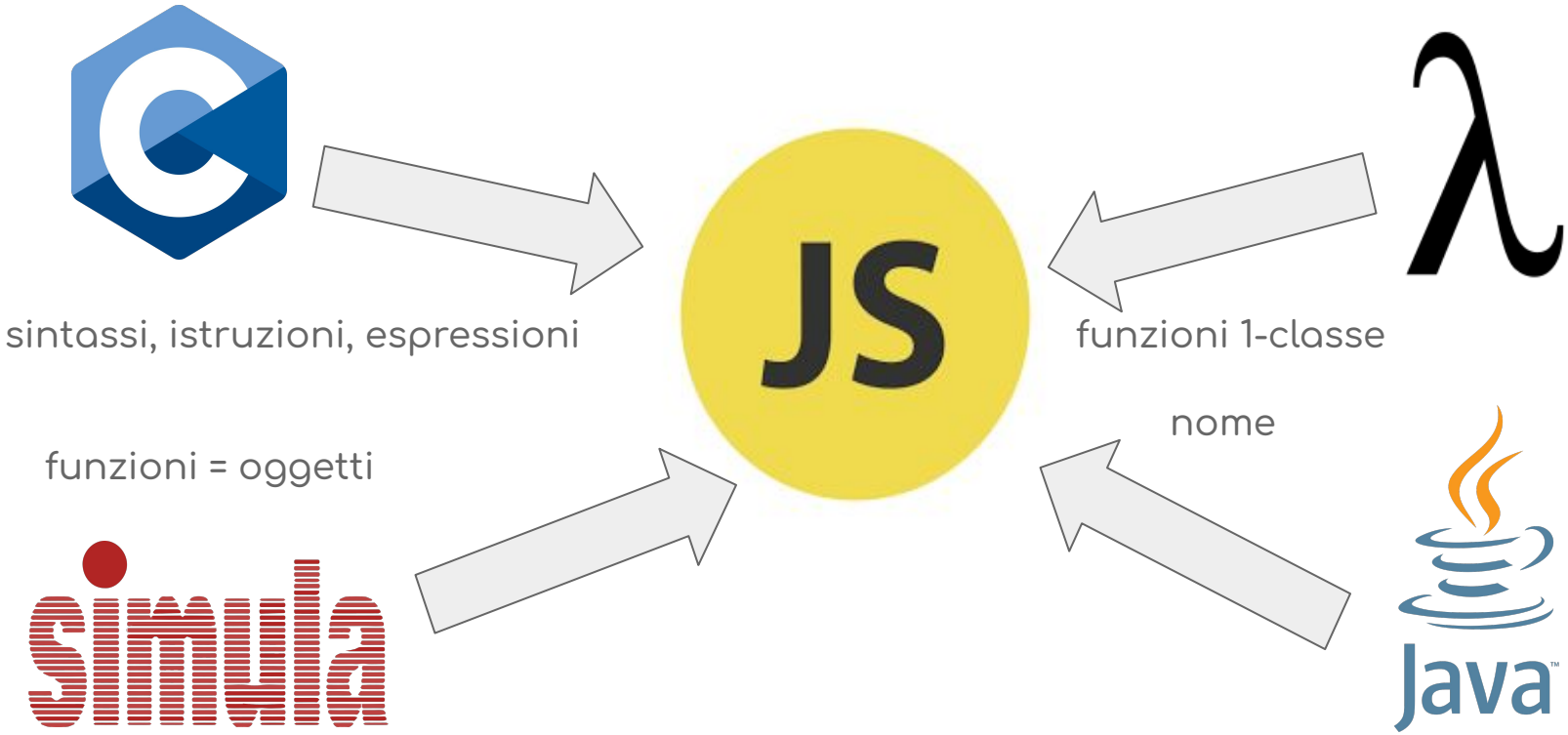
Un linguaggio è elegante se può implementare sé stesso in modo naturale



Javascript

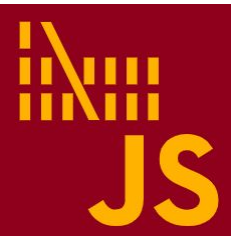


Un crocevia importante



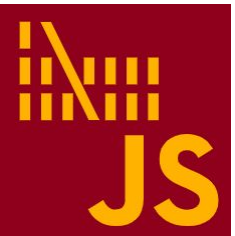
Un sottoinsieme classico di JS

```
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  = prefix_op* value (binary_op expression)?
value       = constant
            | object
            | variable
            | "(" expression ")"
```



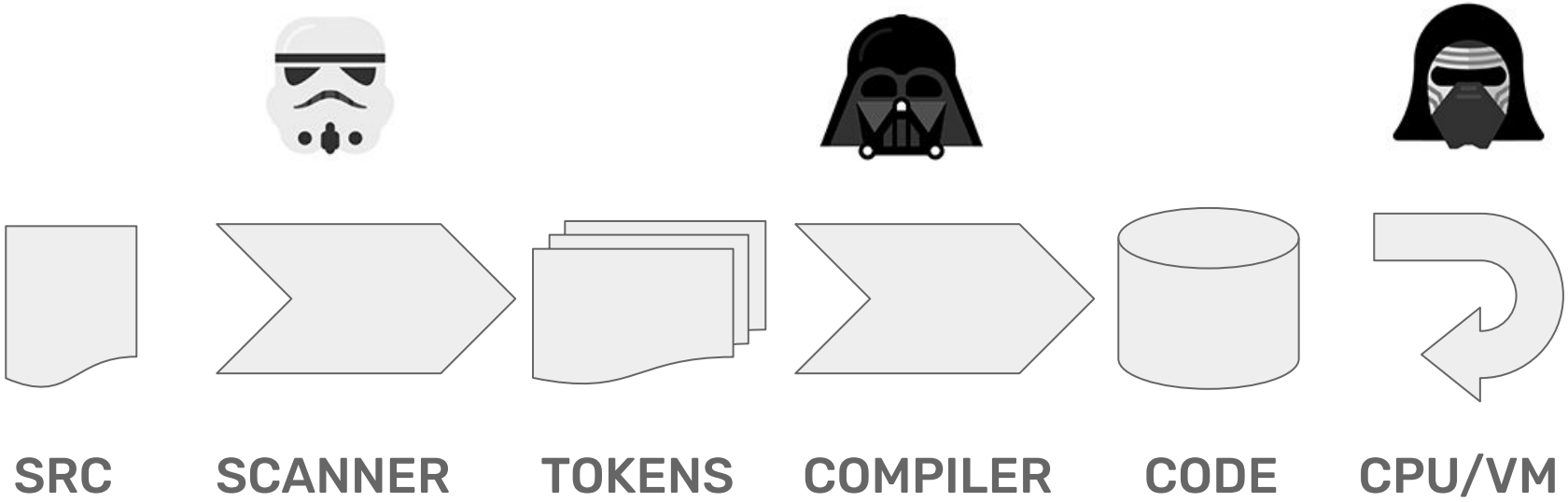
Un sottoinsieme classico di JS

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



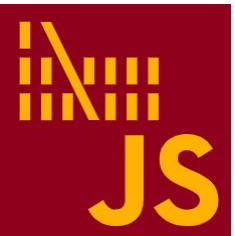
Come si progetta un compilatore?

Un po' facile, un po' difficile



In Javascript è facile :-)

```
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!

compile time vs. run time

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!!!

La macchina virtuale

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

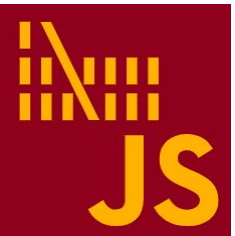
Questa è facile da scrivere una volta definito il set di istruzioni.

Mostra esempio live!

Piccolo elogio delle stack machine

Lo stack è la struttura di dati più fondamentale dell'informatica, inventata da Turing nel 1947 per gestire gli indirizzi di ritorno delle subroutine. Negli anni '60 sono fiorite, per scopi teorici, le macchine virtuali a stack, come quella di Landin per il λ-calcolo o l'interprete per l'Algol-60 di Dijkstra.

Alla fine degli anni '60, le stack machine hanno trovato la massima espressione nei sistemi Forth, inventati da Chuck Moore, ineguagliati per eleganza, semplicità ed efficienza.



Ispiriamoci alla SECD di Landin

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

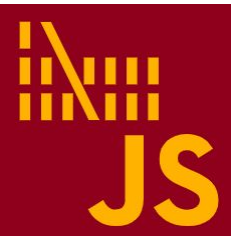
The mechanical evaluation of expressions (1964)

In Javascript è facilissima.

Struttura della SM

La nostra SM possiede i seguenti “registri”, che in realtà sono aree di memoria:

S Stack dove memorizzare valori JS
E Lista di “ambienti” {n1:v1,...} dove memorizzare i valori delle variabili a un certo scope
C Lista con il codice da eseguire
D Dump stack, usato per salvare ambienti nella valutazione delle chiusure

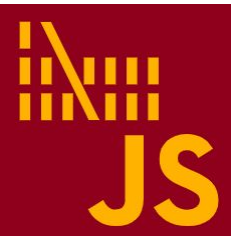


Come funziona la nostra SM

Le istruzioni nella lista C non sono opcode, ma funzioni JS e, occasionalmente, oggetti da premere sullo stack S, che contiene i dati elaborati, quindi le istruzioni non hanno argomenti ma prendono i loro input dallo stack e depositano l'output sullo stack.

Esempio: l'espressione 1 + 2 * 3 viene tradotta in

```
PUSH 1 PUSH 2 PUSH 3 MUL ADD
```



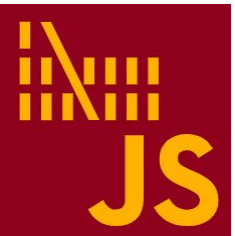
Compilare le istruzioni

Analisi top-down ricorsiva

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;}
```

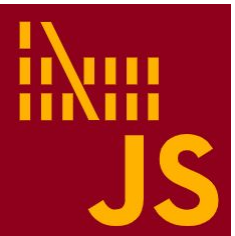
Mostra il codice!!!



Analisi top-down ricorsiva

Il codice che compila una istruzione chiama i codici che compilano le categorie sintattiche (espressioni, variabili, etc.) usare dalle istruzioni in una organizzazione gerarchica di funzioni. Esempio:

```
compila ciclo do-while:
   a = rt.ic
   compila un blocco;
   deve esserci "while" (se no errore);
   compila espressione;
   compila "JPNZ a".
```



Istruzioni di controllo => goto!

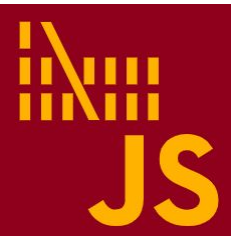
Ogni istruzione di controllo si esprime in termini di combinazioni salti incondizionati (JP) e condizionati (JP, JPNZ).

Mostralo sul compilatore!!!

Compilare le espressioni

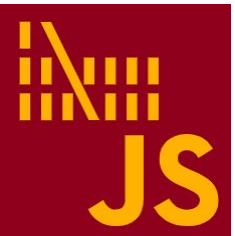
Analisi bottom-up con precedenza

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



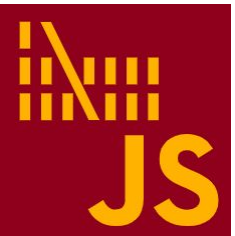
Analisi bottom-up con precedenza

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



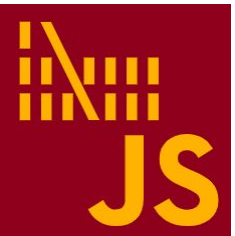
Analisi bottom-up con precedenza

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



Analisi bottom-up con precedenza

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



Analisi bottom-up con precedenza

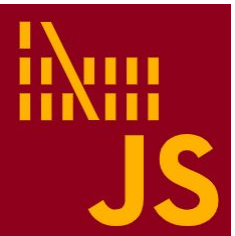
```
1 + x * (x - 1) => c = []        s = []
+ x * (x - 1)   => c = [1]     s = []
x * (x - 1)  => c = [1]     s = [+]
* (x - 1)       => c = [1 x]     s = [+]
(x - 1)         => c = [1 x]     s = [+ *]
```

Analisi bottom-up con precedenza

```
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 = [+ * (]
```

Analisi bottom-up con precedenza

```
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 = [+ * (]
```



Analisi bottom-up con precedenza

```
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 = [+ * ( -]
```

Analisi bottom-up con precedenza

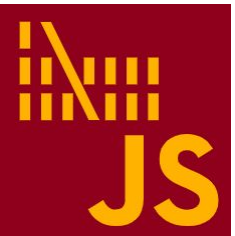
```
1 + x * (x - 1) => c = []          s = []
+ x * (x - 1)   => c = [1]     s = []
x * (x - 1)  => c = [1]     s = [+]
* (x - 1)        => c = [1 x]      s = [+]
(x - 1)          => c = [1 x]      s = [+ *]
x - 1)       => c = [1 x]      s = [+ * (]
- 1)             => c = [1 x x]    s = [+ * (]
1)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

Analisi bottom-up con precedenza

```
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 - * +]
```

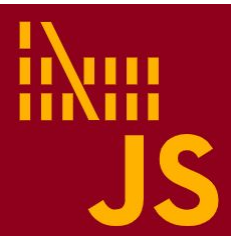
Valutazione in forma post-fissa

```
Supponiamo env = {x:2}

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



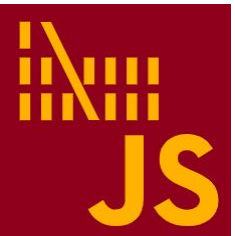
Valutazione in forma post-fissa

```
Supponiamo env = {x:2}

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



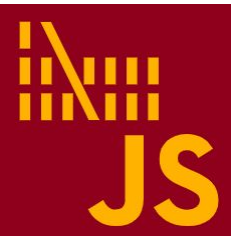
Valutazione in forma post-fissa

```
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]
```



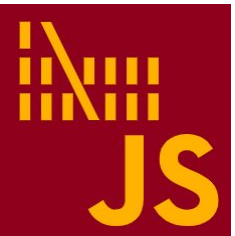
Valutazione in forma post-fissa

```
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]
```



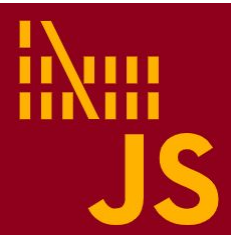
Valutazione in forma post-fissa

```
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]
```



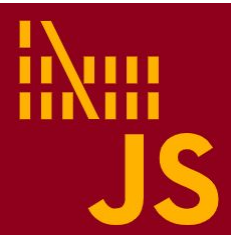
Valutazione in forma post-fissa

```
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 post-fissa

```
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 post-fissa

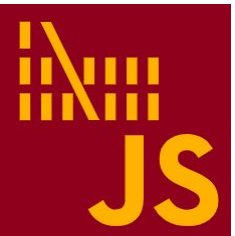
Supponiamo env = {x:2}

```
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)

Grazie per l'attenzione

Q&A?



Paolo Caressa @ GSE spa

https://github.com/pcaressa/

https://linkedin.com/in/paolocaressa

https://twitter.com/www_caressa_it