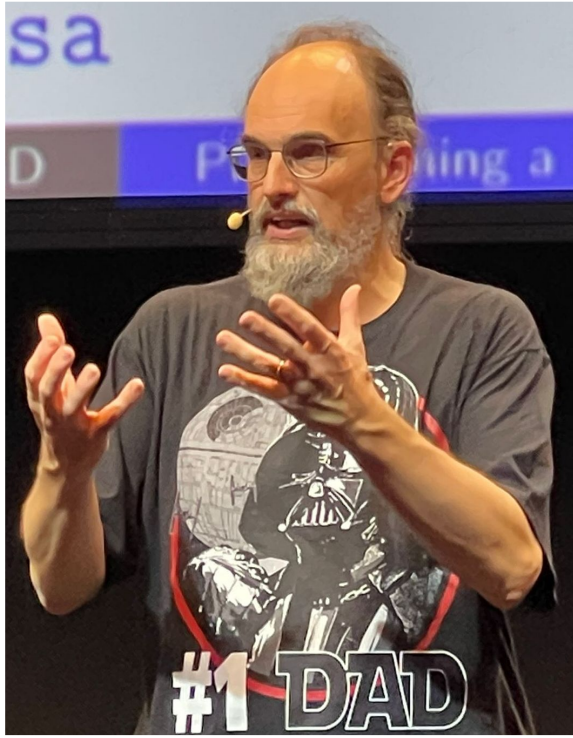
Elegant by Design: JavaScript in JavaScript

Paolo Caressa @GSE spa

Bologna, April 8th, 2025

Milestones in the life of a quiet man



1975 Learn to read from Spider-Man comics.
1977 See Star Wars at the cinema.
1983 Start to program in BASIC on a ZX Spectrum 16K.
1985 Begin to study computer science seriously.
1988 Enroll in Mathematics, to become a computer scientist.
1994 Graduate in algebra, to become a mathematician.
1995 Buy a PC (Win 3.11): instead of Win95 I install Linux on it.
2000 PhD in geometry, to become a researcher.
2001 IT consultant (AI library in C).
2005 Full-Time Quant: Mathematics, C, C++, C#, and Fortran.
2009 From Risk Manager in Java to Project Manager in Excel.
2010 Get married and have a daughter, learn JavaScript for her.
2016 Start to hang out in AI communities.
2019 Hired as IT Expert at GSE.

Currently, I participate in communities, write (books, papers, posts etc.), speak and some people keep reading and listening to me!!!

Why does a boomer like me talk about JavaScript?

I discovered JS in 2013, when trying to develop an app for my daughter: she wanted to draw lines, rectangles and circles. I had a iPhone, writing an app was a nightmare.

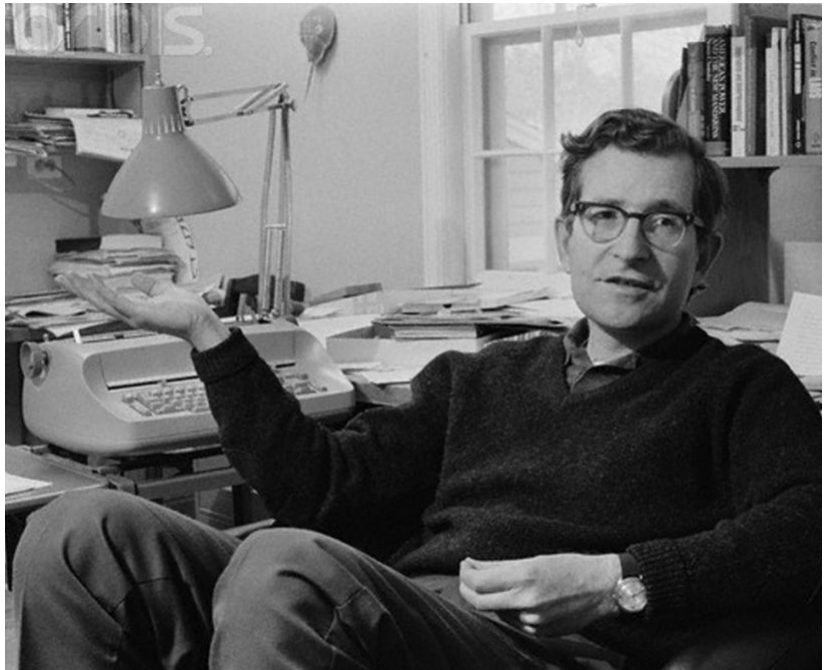
So I decided to use HTML5 + JS and discovered its power and elegance.

Languages, recursion and self-reference

The discreet charm of programming languages

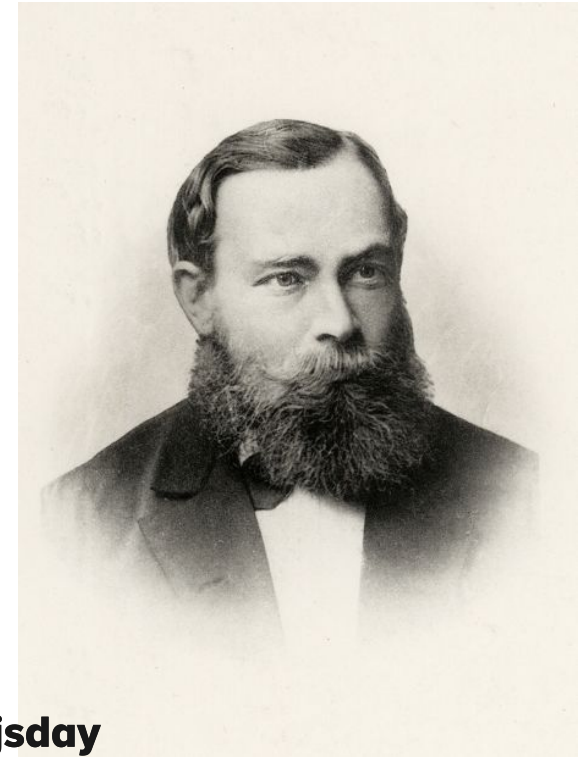
Programming languages are examples of **formal systems**.

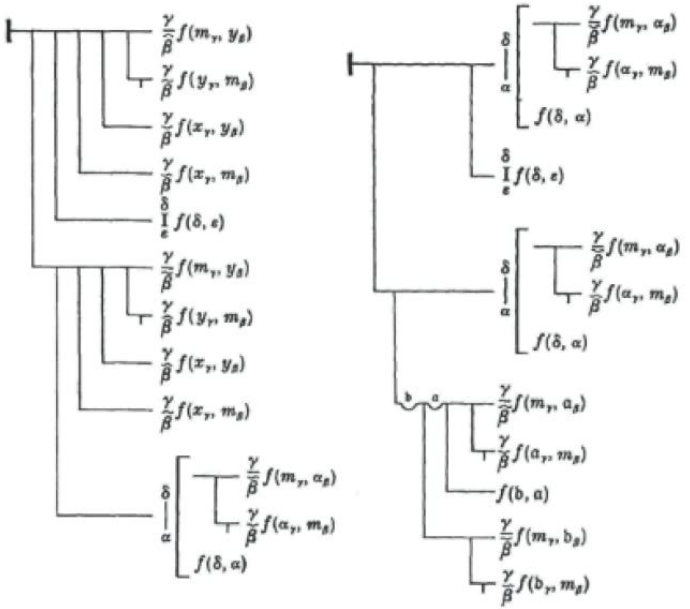
One can **generate** all strings corresponding to correct programs by means of suitable automata.



Noam Chomsky

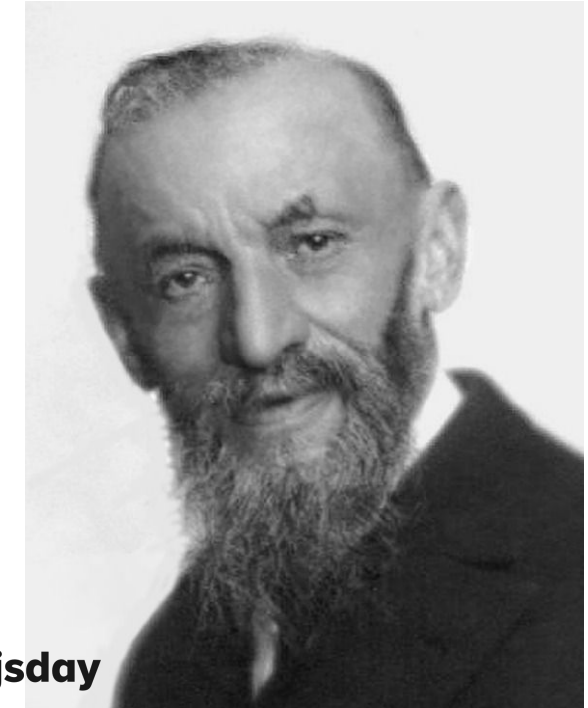
It’s all logicians fault…





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*

The formal language par excellence

Wir geben nun das zugehörige Axiomensystem an. Als logische Grundformeln haben wir zunächst die Axiome des Aussagenkalküls, die wir der Einfachheit halber in derselben Form wie früher geben.

a) $X \vee X \rightarrow X$.

b) $X \rightarrow X \vee Y$.

c) $X \vee Y \rightarrow Y \vee X$.

d) $(X \rightarrow Y) \rightarrow [Z \vee X \rightarrow Z \vee Y]$.

($\mathfrak{A} \rightarrow \mathfrak{B}$ ist hier wieder wie früher als eine Abkürzung für $\bar{\mathfrak{A}} \vee \mathfrak{B}$ aufzufassen.)

Dazu kommen jetzt als zweite Gruppe zwei *Axiome* für *„alle“*, *und „es gibt“* hinzu.

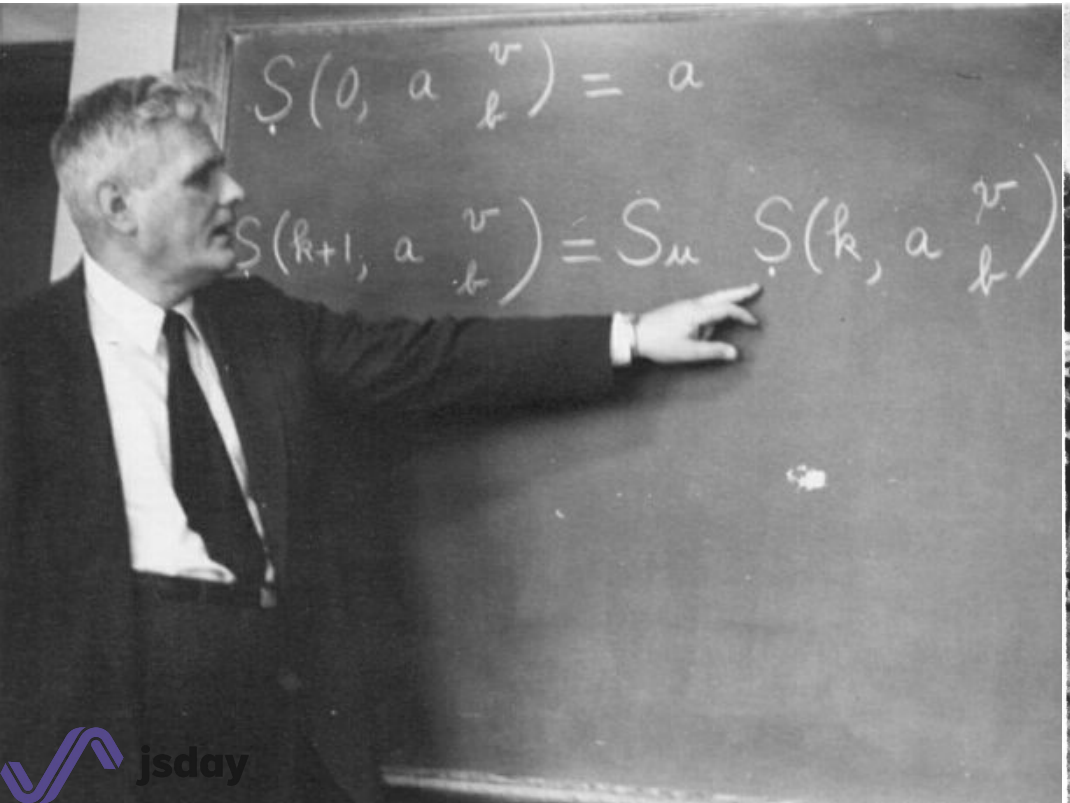
e) $(x)\,F(x) \rightarrow F(y)$.

f) $F(y) \rightarrow (E\,x)\,F(x)$.

David Hilbert *First order predicate calculus*

Algorithms, automata and languages

Alonzo Church



Alan Turing

jsday

The first(?) programming language

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

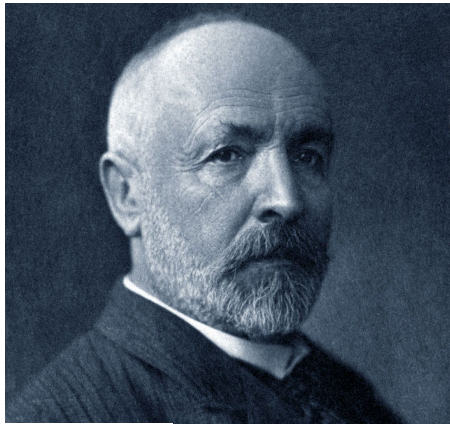
$$
\begin{array}{l|l}
 & W1(m-1)\left[\begin{array}{ll|l} Z & \Rightarrow Z & i \Rightarrow \varepsilon \\ 0 & 1 & \\ i+1 & & \\ (\sigma,\tau) & (\sigma,\tau) & 1.n \quad 1.n \end{array}\right. \\
V & \\
K & \\
S & \\
 & W\left[\varepsilon\geq 0\rightarrow\left[\begin{array}{l} Z{<}Z \ \vee (Z{=}Z \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

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

Konrad Zuse *Plankalkül*

To iterate is human, to recurse is divine

Richard Dedekind



Georg Cantor

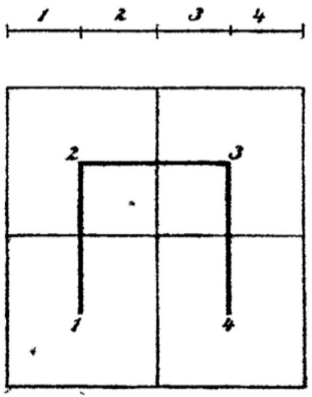


Fig. 1.

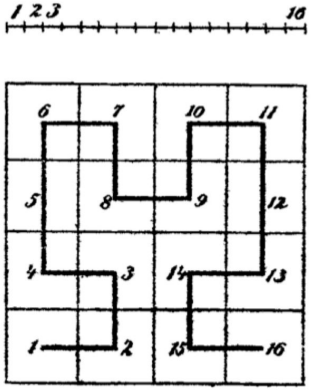


Fig. 2.

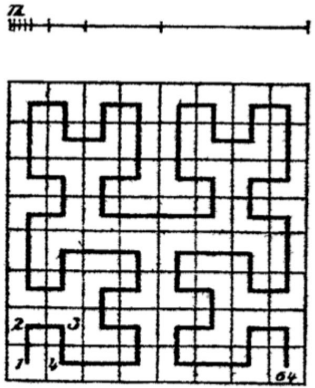


Fig. 3.

jsday

Programming language grammars are recursive

<expr> ::= <value> |
    <value> <opt> <expr>

<value> ::= <num> | <var> | (<expr>)

<num> = <digit> | <num><digit>

<opt> = + | - | * | /

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

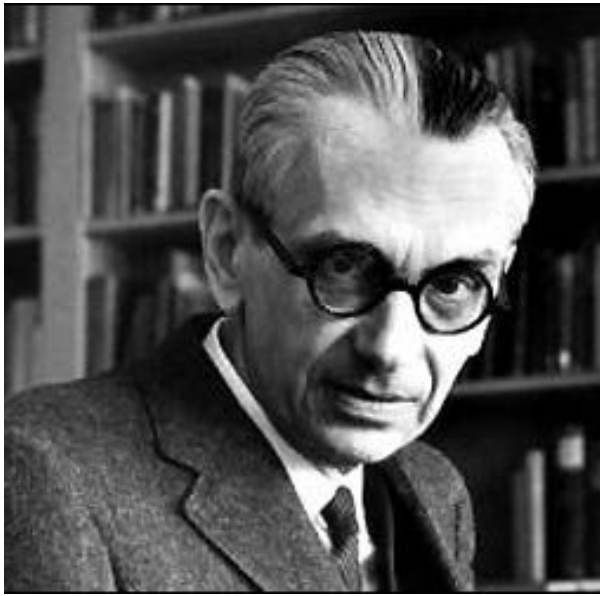
Self-referencing systems

Let $B(n,m)$ be true iff $n = |P|$ encodes a proof P for the formula F encoded by $m = |F|$.

Let $D(n)$ be true iff $\exists n\ B(n,m)$.

Is $D(|\neg D|)$ provable? It asserts “I am not provable” and it cannot be provable, else we could prove both D and $\neg D$. Therefore it is true!

So, if arithmetics is consistent, then there are true but not provable formulas in it.



Kurt Gödel

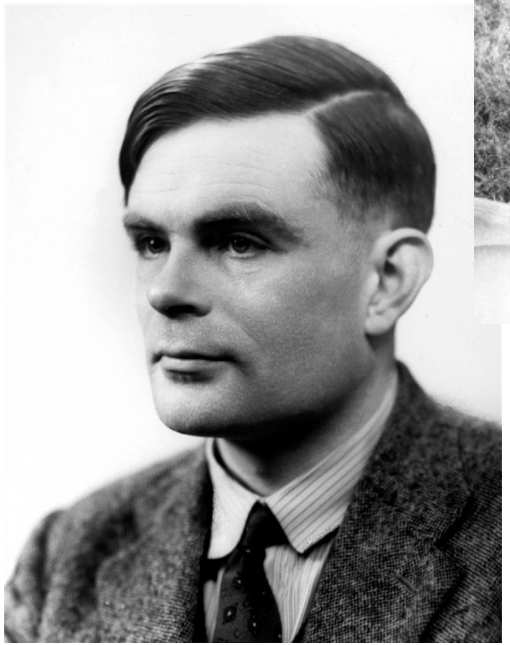
The dangers of self-reference

Let `h(s)` be a JS function which returns true iff the JS `s()` does not fall into an infinite loop.

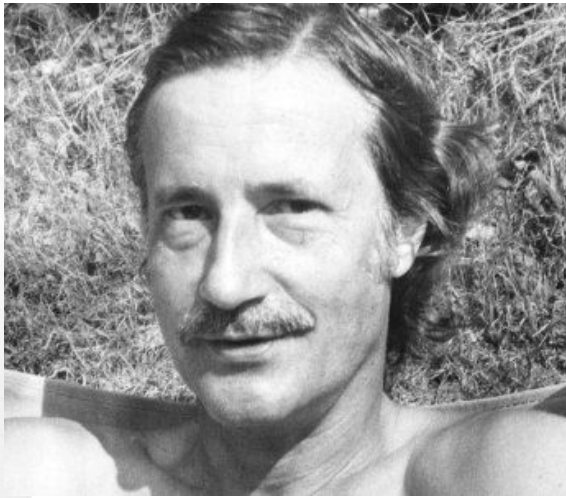
Now consider:

```
function foo() {
    if (h(foo))
        while (true);
}
```

Does it terminate?

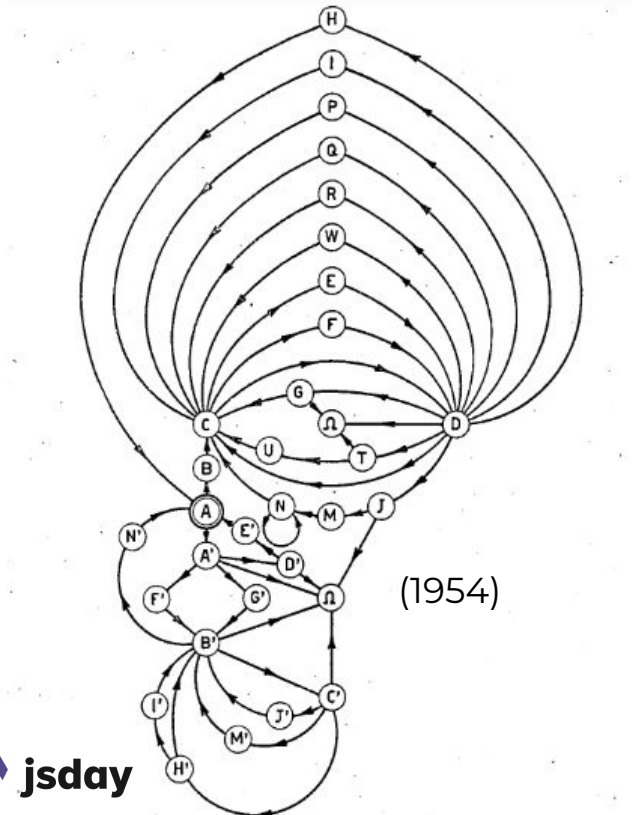


Alan Turing



Christopher Strachey

Corrado Böhm self-interpreter



(1954)

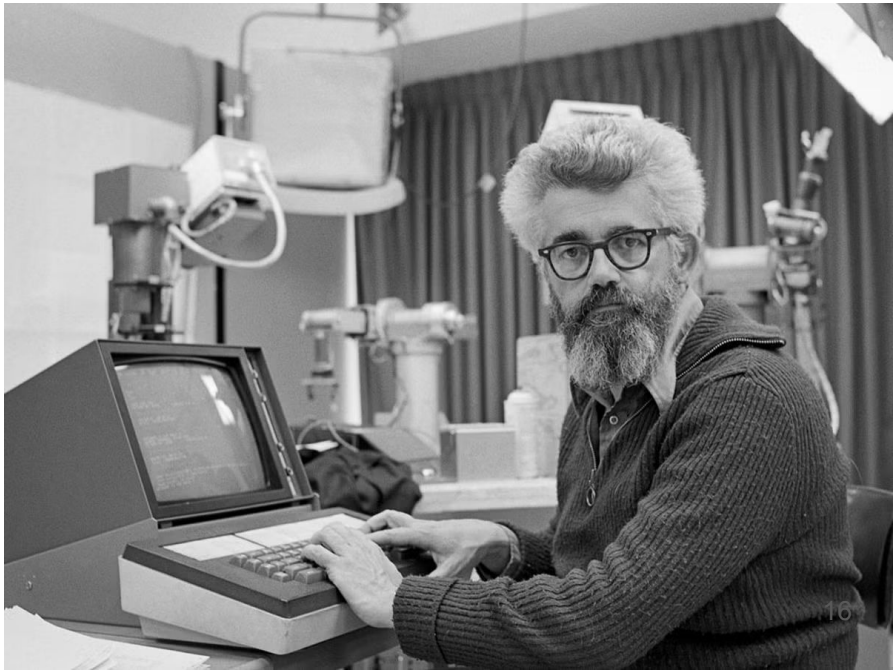
jsday

```
(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) A))))
                  (CDR E)
                  A))
          ((EQ (CAR E) (QUOTE LABEL))
           (EVAL (CONS (CADDR E) (CDR (CDR (CDR E))))
                 (CONS (CONS (CADR E)
                             (CONS 'LAMBDA
                                   (CONS (CADR (CADDR E))
                                         (CONS (CONS (CAR E)
                                                     (CONS (CADR E)
                                                           (CONS (CADDR E) (CADR (CADDR E)))))
                                               NIL)
                                         )))
                       A)))
          (T (EVAL (CONS (CDR (LABEL ASSOC
                                     (LAMBDA (E A)
                                       (COND
                                        ((NULL A) NIL)
                                        ((EQ E (CAAR A)) (CAR A))
                                        (T (ASSOC E (CDR A)))))
                                     (CAR E)
                                     A))
                         (CDR E))
                   A))))
        ((EQ (CAAR E) (QUOTE LAMBDA))
         (EVAL (CADDAR E)
               (LABEL FFAPPEND
                      (LAMBDA (U V)
                        (COND ((NULL U) V)
                              (T (CONS (CAR U)
                                       (FFAPPEND (CDR U) V)))))
                      (LABEL PAIRUP
                             (LAMBDA (U V)
                               (COND ((NULL U) NIL)
                                     (T (CONS (CONS (CAR U) (CAR V))
                                              (PAIRUP (CDR U) (CDR V))))))
                             (CADAR E)
                             (LABEL EVLIS
                                    (LAMBDA (U A)
                                      (COND ((NULL U) NIL)
                                            (T (CONS (EVAL (CAR U) A)
                                                     (EVLIS (CDR U) A)))))
                                    (CDR E)
                                    A))
                      A))))))
```

(1958)

Lisp in Lisp

John McCarthy



jsday

META II in META II

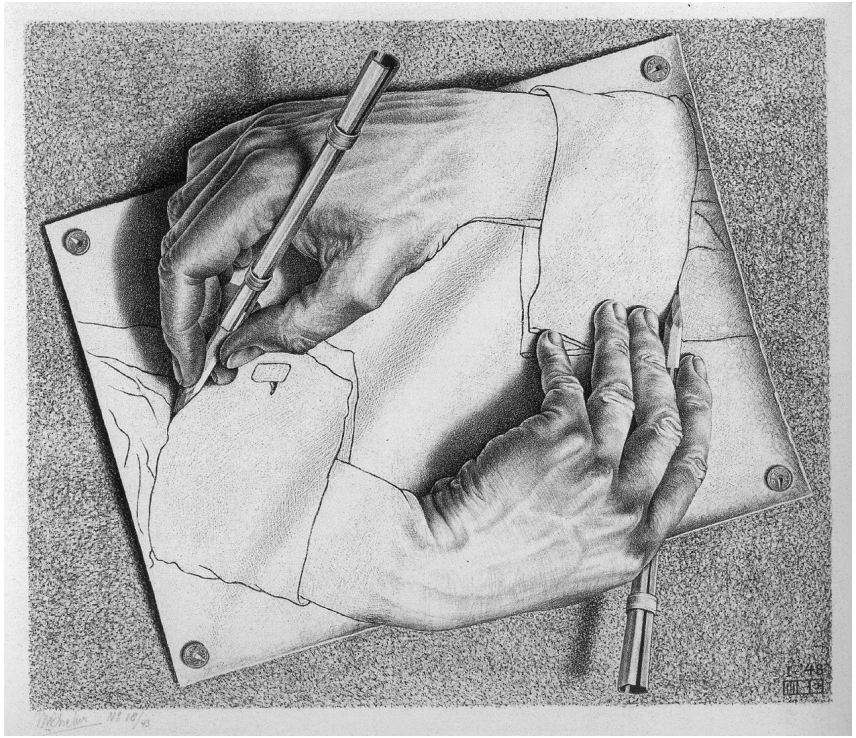
The compiler-compiler META II, can describe itself, here on the right! It translated into a VM machine language.

It was written by Dewey Val Schorre at UCLA in 1964

```
.SYNTAX PROGRAM

OUT1 = '*1' .OUT('GN1') / '*2' .OUT('GN2') /
'*' .OUT('CI') / .STRING .OUT('CL ' *).,

OUTPUT = ('.OUT' '('
$ OUT1 ')' / '.LABEL' .OUT('LB') OUT1) .OUT('OUT') .,

EX3 = .ID .OUT ('CLL' *) / .STRING
.OUT('TST' *) / '.ID' .OUT('ID') /
'.NUMBER' .OUT('NUM') /
'.STRING' .OUT('SR') / '(' EX1 ')' /
'.EMPTY' .OUT('SET') /
'$' .LABEL *1 EX3
.OUT ('BT ' *1) .OUT('SET').,

EX2 = (EX3 .OUT('BF ' *1) / OUTPUT)
$(EX3 .OUT('BE') / OUTPUT)
.LABEL *1 .,

EX1 = EX2 $('/' .OUT('BT ' *1) EX2 )
.LABEL *1 .,

ST = .ID .LABEL * '=' EX1
'.,' .OUT('R').,

PROGRAM = '.SYNTAX' .ID .OUT('ADR' *)
$ ST '.END' .OUT('END').,

.END
```

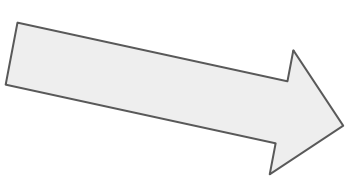
Aesthetic principle

The more a language can implement itself in simple and natural way, the more it is **elegant.**

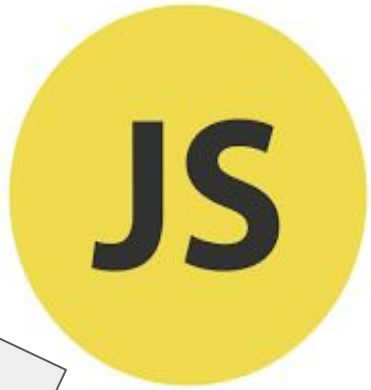


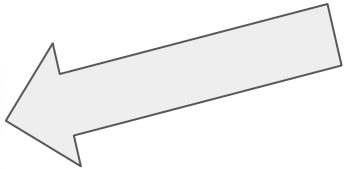
JavaScript

A language at the crossroads



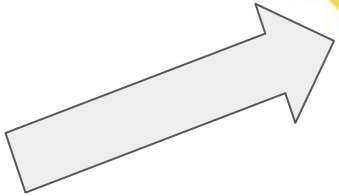
syntax, instructions, expressions



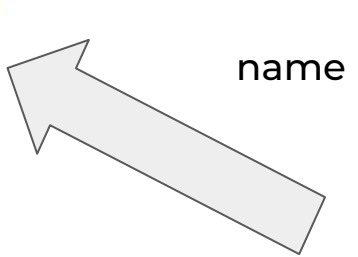


1-class functions

functions = objects



name



A plain vanilla subset of JS

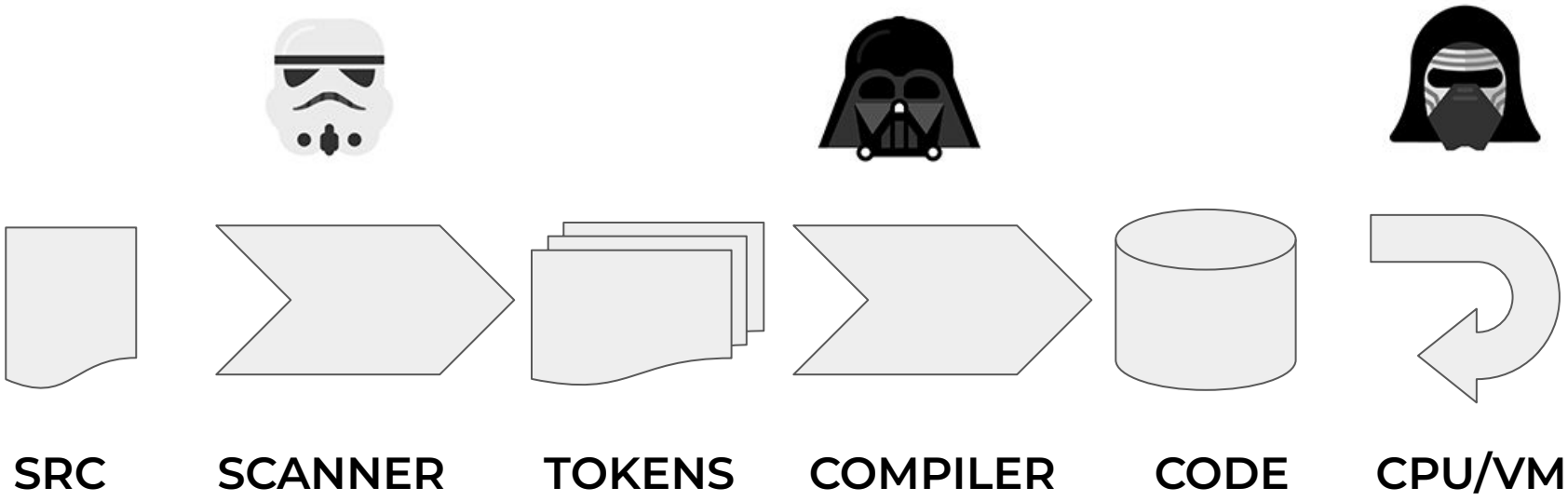
```
program    = statement*

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
```

A plain vanilla subset of JS

```
expression = prefix_op* value (binary_op expression)?

value = constant | object | variable | "(" expression ")"
constant = digit(digit)*("."(digit)*)? | "null" | "undefined"
       | "true" | "false" | "'"(character)*"'" | '"'(character)*'"'

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

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

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

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

How to build a compiler?

Some part easy, some difficult



In JavaScript is easy

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

The scanner

function scan(text): accepts a text and returns a token list, thus an object array.

```
token = {
        s: "string representation",
        t: "type (number, string, ...)",
        l: line number,
       c: column number
}
```

Show it!

The actual compiler

function compile(tl): accepts a token list and returns an object of "runtime" type:

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


Show it!

Don't let any s/w engineer see that

Usually a compiler is engineered at least in two modules: front-end and back-end. The first is aware of the source language, the second is aware of the runtime system.

In this case, to let the compiler be able to compile itself, I've written it in the chosen JS subset as a single monolithic piece of code.

The runtime engine

function run(rt): accepts a runtime object and "executes" it.

This is pretty easy to code once the object code and runtime resources have been defined.

Show it!

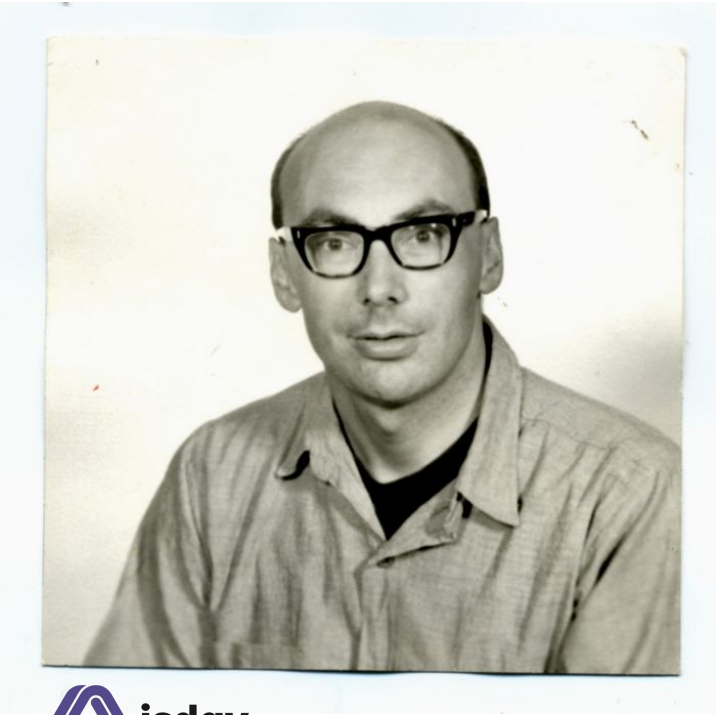
In praise of stack machines

Stack were invented by Turing himself as early as in 1947, to handle subroutine return addresses.

In the 60s, virtual stack interpreters appeared, such as Peter Landin's for λ-calculus or EW Dijkstra's for Algol-60.

Later in the 60s, stack machines found their maximum expression in Chuck Moores' Forth language, still unmatched for elegance, simplicity and efficiency.

Drawing inspiration from SECD



Stack/Environment/Control/Dump machine was proposed by Peter J. Landin in 1963 to describe the operational semantics of λ-calculus, in his paper: *The mechanical evaluation of expressions*.

In Javascript it is easy to implement.

SECD architecture

This VM has four "registers", actually memory areas:

S Stack where values are stored
E A list of environments {name: value,...} where to store variable values at a certain scope
C List to object codes to execute
D Dump stack, used to save environments in closure evaluations (not strictly needed)

How our stack machine works

Instructions in the C list are not opcodes, but actual JS functions and possibly JS objects used as parameters.

Usually, those JS functions take and return parameter from and to the stack S.

So the code is assembled in postfix Polish form.

Example: expression `1+2*3` is compiled as

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

Compiling instructions

Top-down recursive parsing

An instruction starts with a keyword and terminates with ";". Else, it is interpreted as an expression.

```
if (token == "do") {compile do-while;}
elif (token == "if") {compile if-else;}
elif (token == "let") {compile assignment;}
   ...
else {compile an expression;}
```

Show the code!!!

Example: loop compilation

Instruction parsers call the functions that parse and compile other syntactic categories (expressions, variables, etc.): each category is implemented by a function. Example

```
compile do-while:
    a = rt.ic
    compile a block;
    "while" needed (else error);
    compile expression;
    compile "JPNZ a".
```

Control instructions = gotos!

Each control instruction (**if**, **while**, **for**, ...) is expressed in terms of combined jumps, either unconditioned (JP) or conditioned (JPZ, JPNZ, ...)

`Show that on the code!`

Compiling expressions

Precedence bottom-up parsing

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

Precedence bottom-up parsing

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

Precedence bottom-up parsing

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

Precedence bottom-up parsing

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

Precedence bottom-up parsing

```
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 = [+ *]
```

Precedence bottom-up parsing

```
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 = [+ * ()
```

Precedence bottom-up parsing

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

Precedence bottom-up parsing

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

Precedence bottom-up parsing

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

Precedence bottom-up parsing

```
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 - * +]
```

Now evaluate is easy

Let us suppose that **`env = {x:2}`**

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

Evaluating post-fixed form

Let us suppose that **`env = {x:2}`**

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

Evaluating post-fixed form

Let us suppose that **`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]
```

Evaluating post-fixed form

Let us suppose that **`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]
```

Evaluating post-fixed form

Let us suppose that **`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]
```

Evaluating post-fixed form

Let us suppose that **`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]
```

Evaluating post-fixed form

Let us suppose that **`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]
```

Evaluating post-fixed form

Let us suppose that **`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]
```

The result 3 is on the top of the stack.

Live samples

Thanks!
Q&A

Paolo Caressa, PhD
GSE spa
M&MoCS
https://github.com/pcaressa/
https://linkedin.com/in/paolocaressa
https://bsky.app/profile/pcaressa.bsky.social
https://twitter.com/www_caressa_it

(this is a self referencing presentation)