



REFLEXIÓ

REFLEXIÓ

Jordi Delgado (jdelgado@cs.upc.edu)

Reflection: *the ability of a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of an object at runtime*

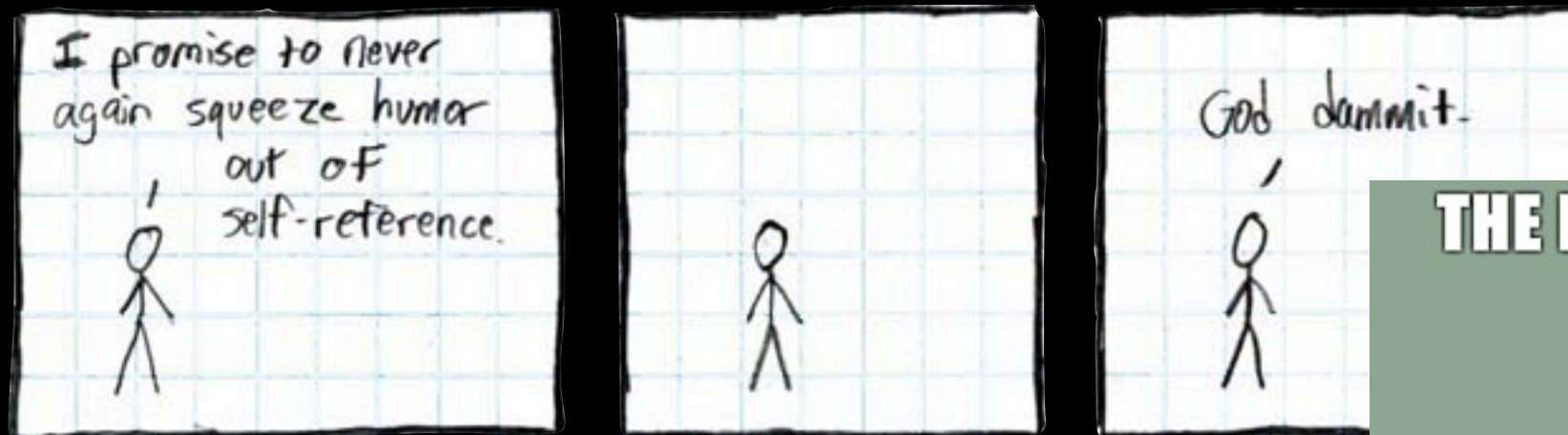
[http://en.wikipedia.org/wiki/Reflection_\(computer_programming\)](http://en.wikipedia.org/wiki/Reflection_(computer_programming))

És l'estudi de (les conseqüències de)
l'**auto-referència**
en els llenguatges de programació.



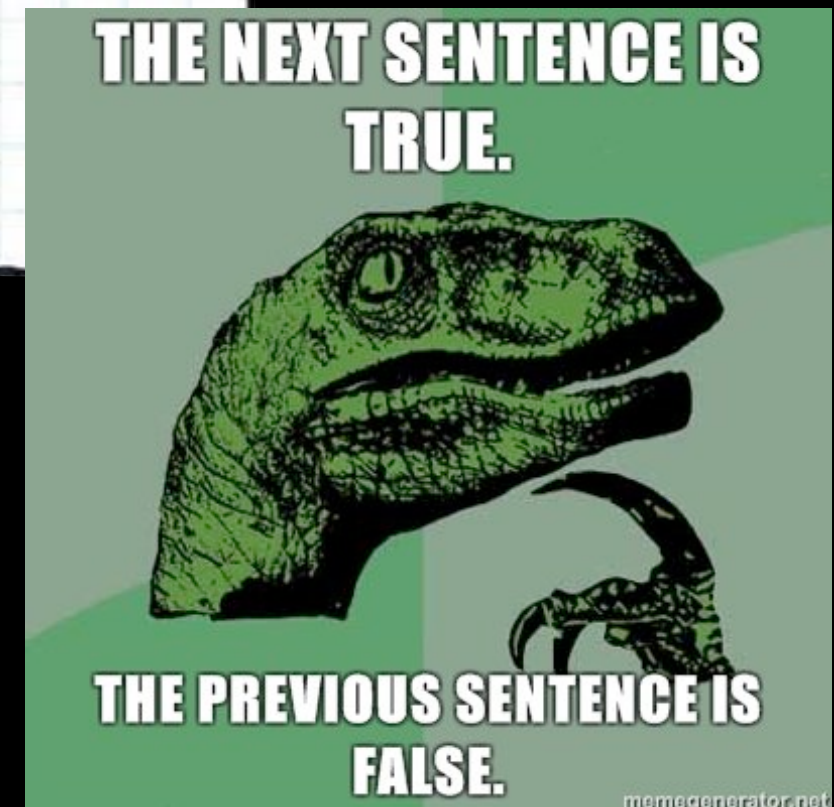
L'auto-referència en el llenguatge té sovint conseqüències paradòxiques i divertides

Hi ha tres tipus de persones al món, els que saben comptar i els que no

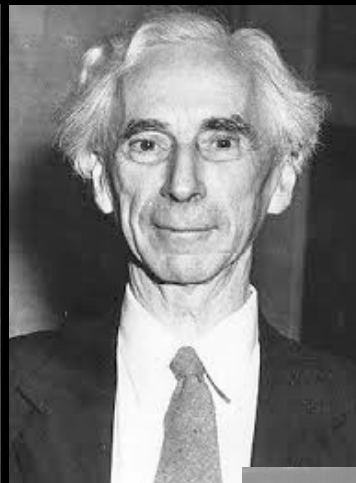
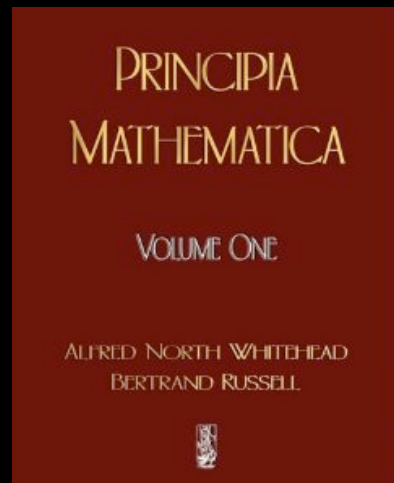


<http://xkcd.com/33/>

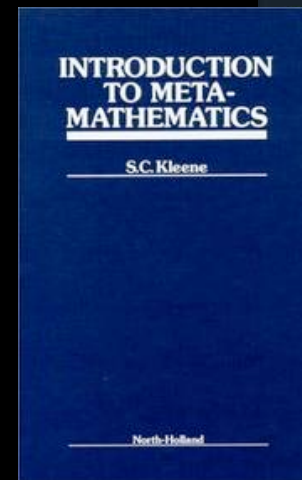
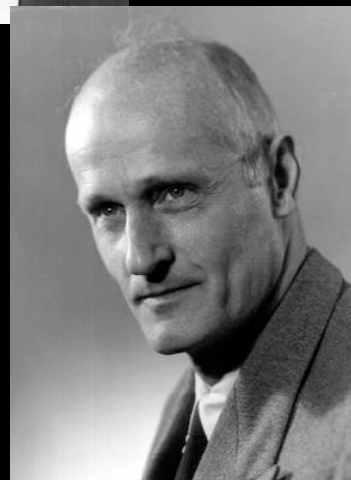
Recursive loop: See recursive loop.
[from Borland Pascal with Objects
7.0: Language Guide (1992);



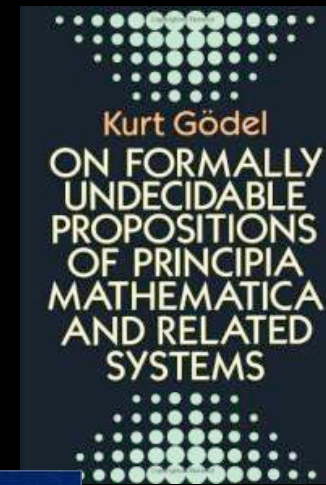
És força conegut l'impacte que ha tingut en matemàtiques el reflexionar al voltant de l'auto-referència



Bertrand Russell, 1872-1970
Alfred N. Whitehead, 1861-1947
Principia Mathematica



Stephen Cole Kleene,
1909-1994
Teoremes de Recursió



Kurt Gödel, 1906-1978
Teoremes d'Incompletesa

Fins i tot en el cinema...



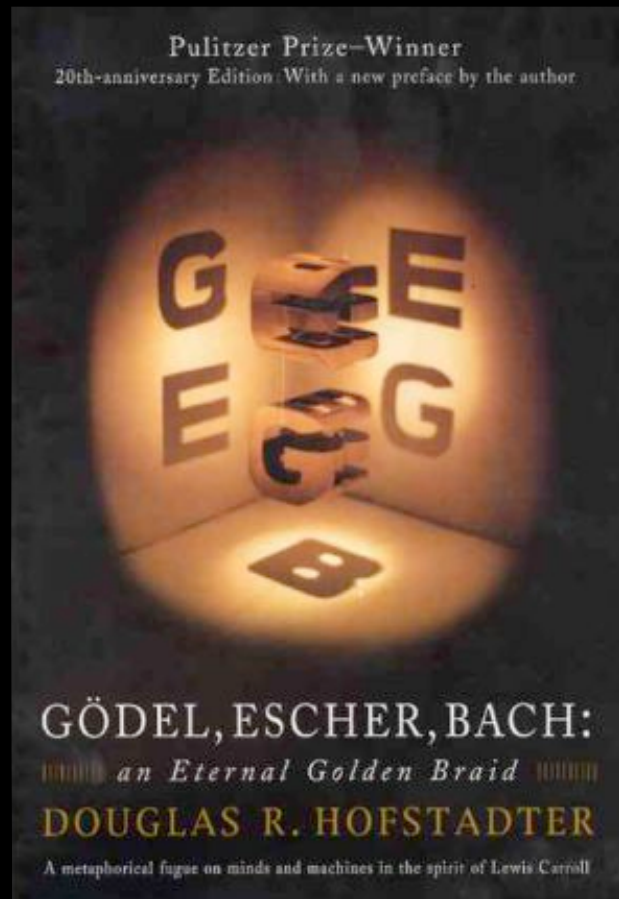
Animal Crackers, 1930
(*El Conflicto de los Marx*)



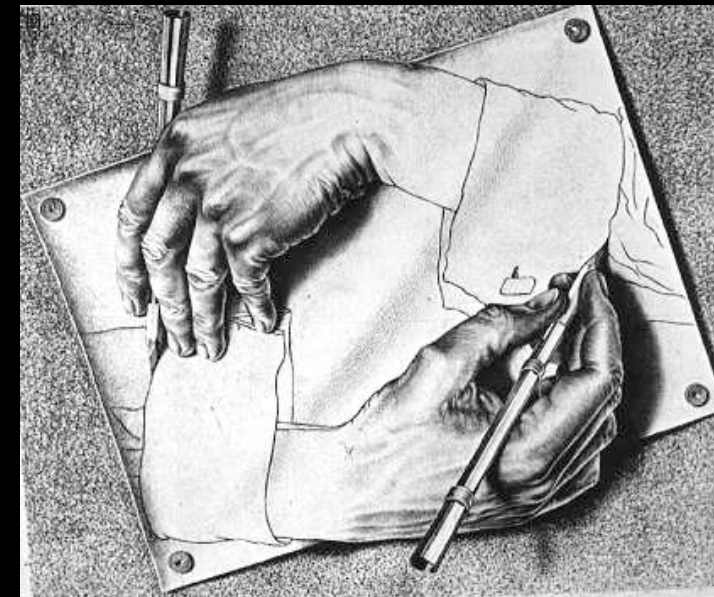
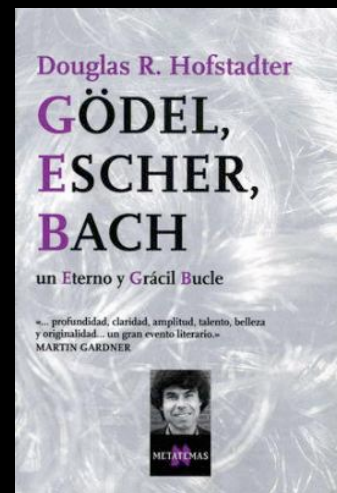
Dumb-Hounded
Tex Avery, MGM 1943



Fight Club, 1999
(*El Club de la Lucha*)



Gödel, Escher, Bach: An Eternal Golden Braid Douglas R. Hofstadter (1979)

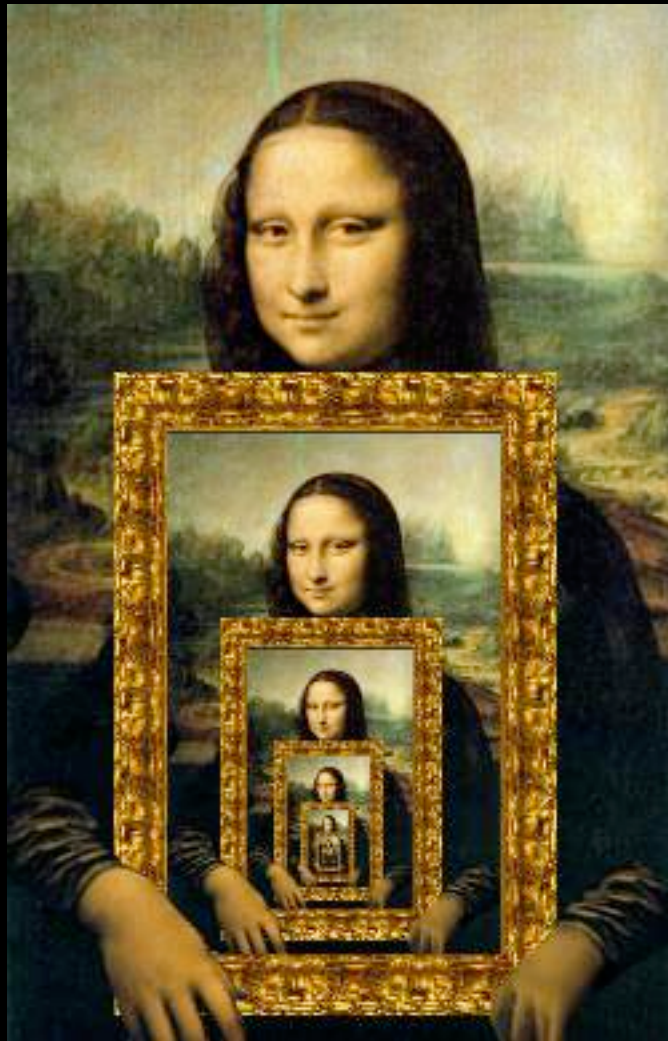


Auto-referència: Com és possible que un sistema sigui capaç de pensar-se a si mateix?



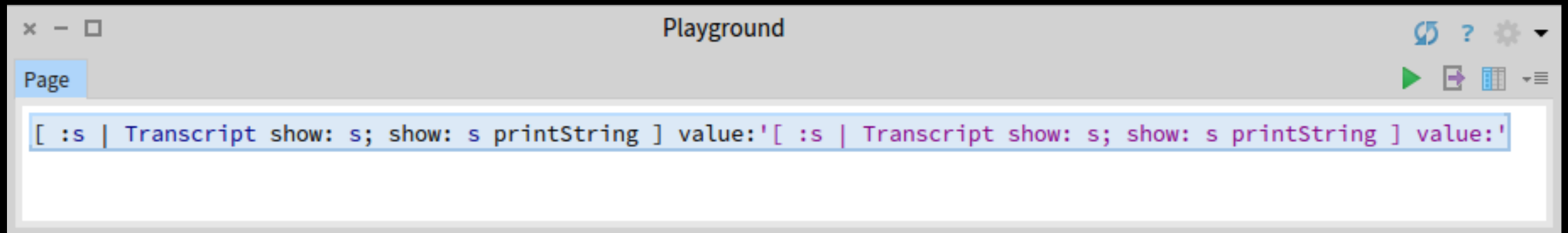
A l'estiu del 2007 es va fer un curs al MIT sobre GEB
<http://ocw.mit.edu/high-school/courses/godel-escher-bach/>

En informàtica hi ha moltes *oportunitats* per a l'*auto-referència*.

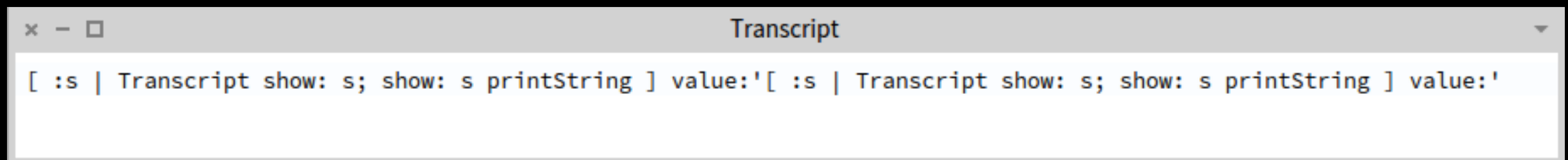


Una de ben coneguda per vosaltres és la propietat d'una funció (mètode, procediment, etc.) de cridar-se a ella mateixa, és a dir, la *recursivitat*, i les estructures de dades recursives, però n'hi ha d'altres...

En informàtica hi ha moltes *oportunitats* per a l'*auto-referència*.



A screenshot of a web-based code editor window titled "Playground". The window has a light gray header with standard window controls (close, minimize, maximize) on the left and a toolbar with icons for undo, redo, search, and settings on the right. Below the header, there is a tab labeled "Page". The main area of the window contains a single line of code: `[:s | Transcript show: s; show: s printString] value:'[:s | Transcript show: s; show: s printString] value:'`. The code is color-coded: `[:s |` is blue, `Transcript show: s;` is purple, `show: s printString]` is green, and `value:'` is red.



A screenshot of a web-based transcript window titled "Transcript". The window has a light gray header with standard window controls (close, minimize, maximize) on the left and a dropdown arrow on the right. The main area of the window displays the output of the code from the playground window above: `[:s | Transcript show: s; show: s printString] value:'[:s | Transcript show: s; show: s printString] value:'`. The text is color-coded to match the code in the playground window.

Els programes que s'escriuen a ells mateixos, és a dir, el seu output és el seu mateix llistat. També es coneixen com a *quines*

En informàtica hi ha moltes *oportunitats* per a l'*auto-referència*.

```
public class Quine
{
    public static void main( String[] args )
    {
        char q = 34;          // Quotation mark character
        String[] l = {        // Array of source code
            "public class Quine",
            "{",
            "    public static void main( String[] args )",
            "    {",
            "        char q = 34;          // Quotation mark character",
            "        String[] l = {        // Array of source code",
            "        ",
            "        }",
            "        for( int i = 0; i < 6; i++ )          // Print opening code",
            "            System.out.println( l[i] );",
            "        for( int i = 0; i < l.length; i++ )      // Print string array",
            "            System.out.println( l[6] + q + l[i] + q + ',' );",
            "        for( int i = 7; i < l.length; i++ )      // Print this code",
            "            System.out.println( l[i] );",
            "    }",
            "}",
            "};",
            "for( int i = 0; i < 6; i++ )          // Print opening code",
            "    System.out.println( l[i] );",
            "for( int i = 0; i < l.length; i++ )      // Print string array",
            "    System.out.println( l[6] + q + l[i] + q + ',' );",
            "for( int i = 7; i < l.length; i++ )      // Print this code",
            "    System.out.println( l[i] );",
        };
    }
}
```

Els programes que s'escriuen a ells mateixos, és a dir, el seu output és el seu mateix llistat. També es coneixen com a *quines*

En informàtica hi ha moltes *oportunitats* per a l'*auto-referència*.

Problema de l'aturada:

Teorema

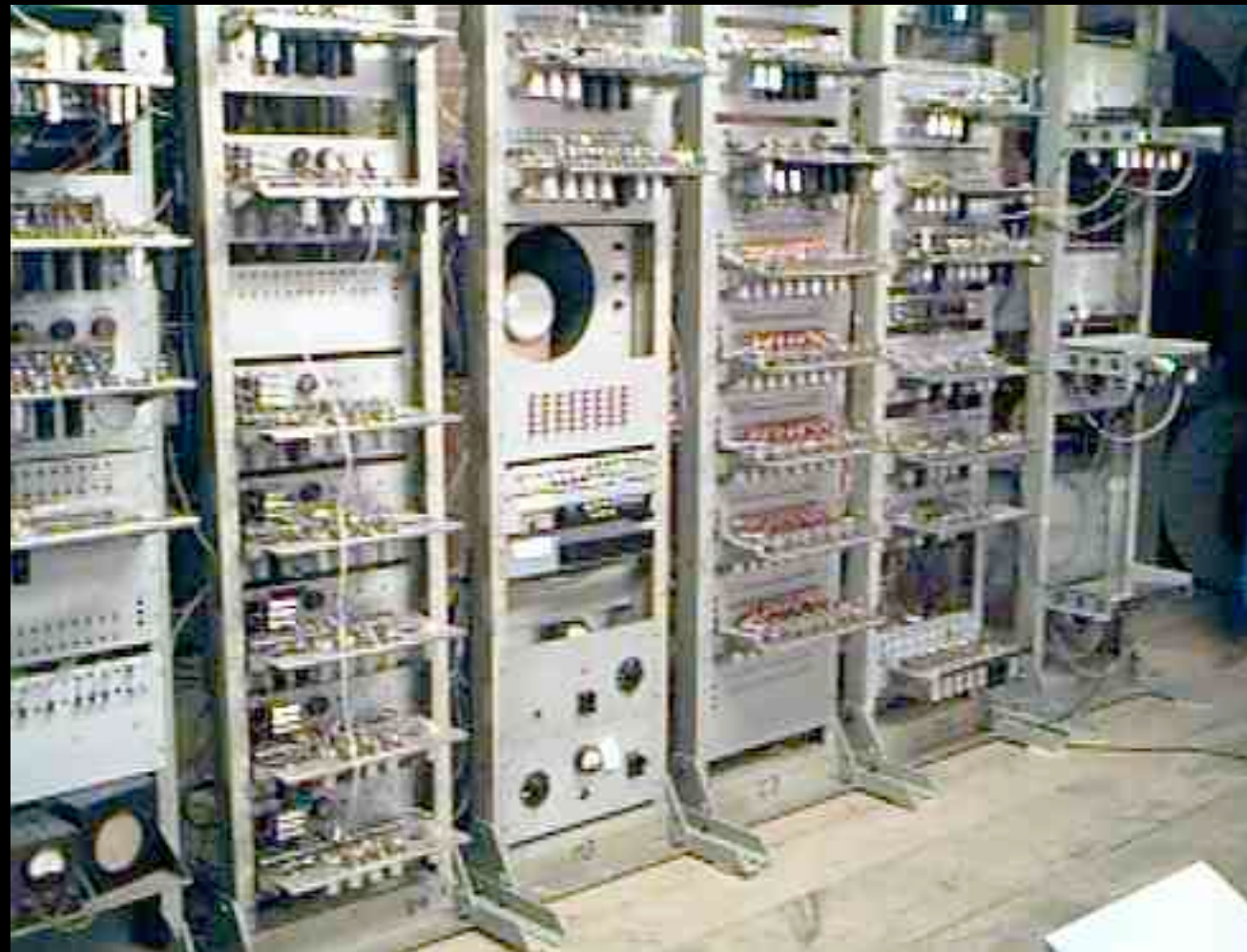
No hi ha cap programa que calculi la funció:

$$\text{halt}(p, d) = \begin{cases} \text{true} & \text{si } p(d) \downarrow \\ \text{false} & \text{si } p(d) \uparrow \end{cases}$$

Idea de la demostració: Suposem que el programa existeix i fent servir l'auto-referència arribem a una contradicció.

Notació: El programa p amb input d s'atura $\equiv p(d) \downarrow$ i direm que $p(d) \uparrow$ si no s'atura.

Però el ventall de possibilitats és més ampli:
Programes que s'*auto-modifiquen*



Manchester Small Scale Experimental Machine (1948)

Simulador: <http://www.davidsharp.com/baby/index.html>

The world's first stored-program computer

Però el ventall de possibilitats és més ampli:
 Programes que *s'auto-modifiquen*

Binary code	Original notation	Modern mnemonic	Operation
000	S, CI	JMP S	Jump to the instruction at the address obtained from the specified memory address S ^[t 1] (absolute unconditional jump)
100	Add S, CI	JRP S	Jump to the instruction at the program counter plus (+) the relative value obtained from the specified memory address S ^[t 1] (relative unconditional jump)
010	-S, C	LDN S	Take the number from the specified memory address S, negate it, and load it into the accumulator
110	c, S	STO S	Store the number in the accumulator to the specified memory address S
001 or 101 ^[t 2]	SUB S	SUB S	Subtract the number at the specified memory address S from the value in accumulator, and store the result in the accumulator
011	Test	CMP	Skip next instruction if the accumulator contains a negative value
111	Stop	STP	Stop

1. ^{a b} As the program counter was incremented at the end of the decoding process, the stored address had to be the target address -1.
2. ^a The function bits were only partially decoded, to save on logic elements.^[26]

Però el ventall de possibilitats és més ampli:
Programes que s'auto-modifiquen

```
0000:00000000000000000000000000000000
0001:100110111111000101111110000111111
0002:00111000100001100000000000000000
0003:00111000100000100000000000000000
0004:10111000100000010000000000000000
0005:10011000000001100000110000110000
0006:11111001110000100000000000000000
0007:11011000100000010000000000000000
0008:00000000100000110000000000000000
0009:01011001110000000011110000111100
0010:10100000000000100000000000000000
0011:11011010001000010000000000000000
0012:00111011011001100000000000000000
0013:00111010101000100000110000110000
0014:10100010001001100000000000000000
0015:10000000000000100000000000000000
0016:11011011111000010000000000000000
0017:00111011110001100011110000111100
0018:00111011111000100000000000000000
0019:10000000000001100000000000000000
0020:01111011110000010000000000000000
0021:00000010001000110000110000110000
0022:00000011110001110000000000000000
0023:00000010010000000000000000000000
0024:00000010001000000000000000000000
0025:00000000000000000111110000111111
0026:01100000000000000000000000000000
0027:11111111111111111111111111111111
0028:0000000000000000000000001000111111
0029:00000000000000000000000000000000
0030:01111011111000101111110000111111
0031:00001011011000000000000000000000

0000: NUM 0 ; ignored (used as a constant)
0001: LDN 25 ; read value from line 25, negate it
0002: STO 28 ; store this value into 28
0003: LDN 28 ; load and negate (i.e +ve value at line 25)
0004: SUB 29 ; subtract value from line 29 (effectively + constant)
0005: STO 25 ; store as new value at line 25 (24, 23...)
0006: LDN 31 ; load value at line 31, negate it
; delay loop follows
0007: SUB 27 ; subtract -1 (add 1)
0008: CMP ; skip jump if negative (initially +ve)
0009: JMP 26 ; jump to line 7
; end of delay loop
0010: LDN 05 ; calculate Line5 minus 1
0011: SUB 27
0012: STO 28
0013: LDN 28
0014: STO 05 ; store -> Line5
0015: LDN 01 ; Line1-1 -> Line1
0016: SUB 27
0017: STO 28
0018: LDN 28
0019: STO 01
0020: SUB 30 ; value at line 30 is to make accumulator +ve when it reaches top
0021: CMP ; skip 'STOP' if still -ve
0022: STOP ; program end
0023: JMP 00 ; jump to value at line0, which is 0, so line 1
0024:
0025:
0026: NUM 6 ; represents line 7
0027: NUM -1
0028:
0029:
0030:
0031:
```

Però el ventall de possibilitats és més ampli:
Programes que s'*auto-modifiquen*

PROGRAM ANALYSIS BY DIGITAL COMPUTER
 by
 DANIEL UNDERWOOD WILDE
 B.S.E.E., University of Illinois
 1961
 S.M., Massachusetts Institute of Technology
 1962
 SUBMITTED IN PARTIAL FULFILLMENT OF THE
 REQUIREMENTS FOR THE DEGREE OF
 DOCTOR OF PHILOSOPHY
 at the
 MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 June, 1966

*Journal of Universal Computer Science, vol. 2, no. 11 (1996), 756-768
 submitted: 13/5/96, accepted: 10/10/96, appeared: 28/11/96 © Springer Pub. Co.*

**Conditional Branching is not Necessary for Universal
 Computation in von Neumann Computers**

Raúl Rojas
 (University of Halle
 Department of Mathematics and Computer Science
 rojas@informatik.uni-halle.de)

**Random-Access Stored-Program Machines, an Approach
 to Programming Languages**

CALVIN C. ELGOT AND ABRAHAM ROBINSON*

IBM Watson Research Center, Yorktown Heights, New York

Journal of the Association for Computing Machinery, Vol. 11, No. 4 (October, 1964), pp. 365-399

**Computational Complexity of Random Access Stored Program
 Machines***

by
 J. HARTMANIS
 Department of Computer Science
 Cornell University
 Ithaca, New York

THE CLASSIC WORK
 NEWLY UPDATED AND REVISED

**The Art of
 Computer
 Programming**

VOLUME I
 Fundamental Algorithms
 Third Edition

DONALD E. KNUTH

Però el ventall de possibilitats és més ampli:
Programes que s'*auto-modifiquen*

L'auto-modificació pot ser necessària per:

- Ofuscació i camuflatge
(virus polimòrfics, *shellcodes*, etc.)
- Optimització (*Just-in-Time compilation*)
- Generació de codi en temps d'execució
(*Synthesis Kernel*)
- Turing-completesa en conjunts limitats d'instruccions
(*OISC machine*)
- Aprenentatge computacional (*Meta-learning*)
- Tolerància a errors (*Space shuttle OS*)



A mida que un llenguatge de programació esdevé *d'un nivell més alt d'abstracció*, la seva implementació en termes de la màquina subjacent comporta *més i més compromisos* per part de l'implementador: Quins casos optimitzar i quins no, etc. L'*habilitat per integrar* quelcom fora de l'abast del llenguatge *es torna més i més limitada*

Gregor Kiczales

Reflexió: Habilitat d'un programa de *manipular com a dades* quelcom que representa l'*estat del programa* mentre aquest s'executa. Hi ha dos aspectes d'aquesta manipulació: *introspecció* i *intercessió*.

Introspecció: Habilitat d'un programa d'*observar*, i per tant *raonar*, sobre el seu propi estat.

Intercessió: habilitat d'un programa de *modificar* el seu propi estat o d'*alterar la seva pròpia interpretació* (semàntica).

Tots dos aspectes requereixen d'un mecanisme per codificar l'estat de l'execució com a dades. Proporcionar aquest estat s'anomena *reificació*.

Bobrow, Gabriel & White, 1993

“Un sistema que es té a ell mateix com a domini (de l’aplicació) i que està *connectat causalment* amb aquest domini pot ser considerat un **sistema reflexiu**”

Pattie Maes, OOPSLA 1987

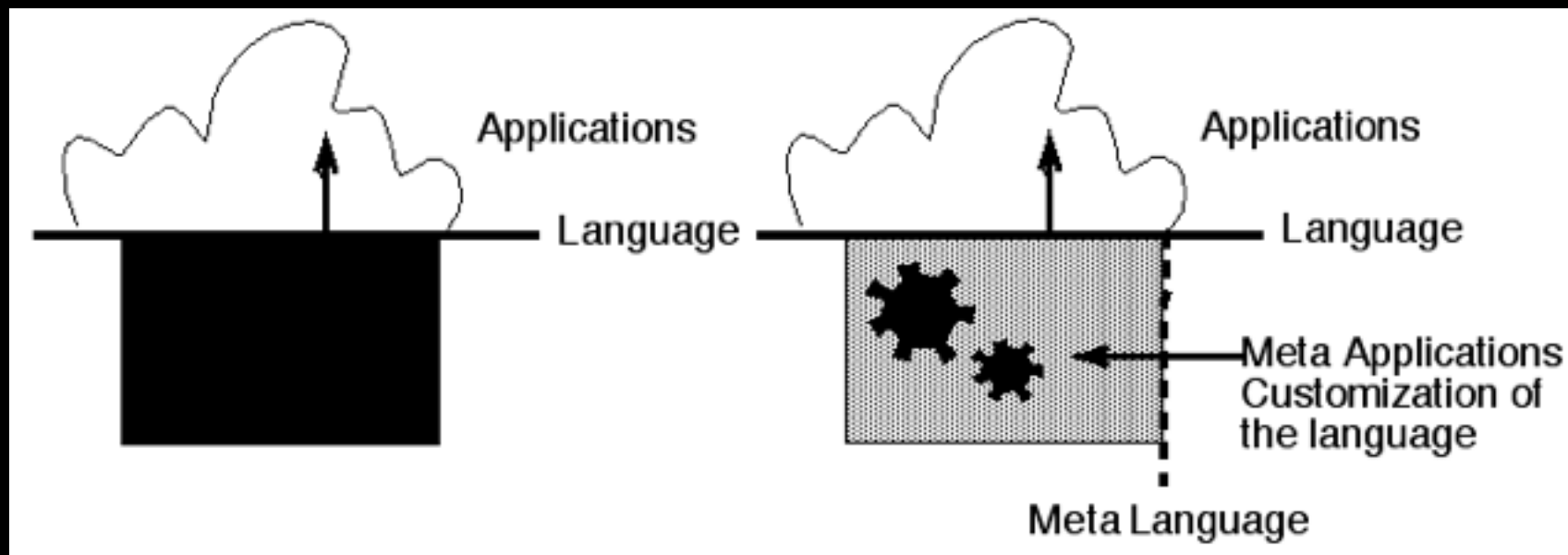
Un **sistema reflexiu** té una *representació interna d’ell mateix*.

Un **sistema reflexiu** és capaç d’*actuar sobre ell mateix* amb la seguretat que aquesta representació estarà connectada causalment amb el sistema.

Un **sistema reflexiu** té capacitat d’*auto-representació* -estàtica- i d’*auto-modificació* -dinàmica- continuament sincronitzades.

El *meta-llenguatge* i el *llenguatge* poden ser:

- diferents (p.ex. Scheme i un llenguatge OO)
- el mateix (p.ex. Smalltalk, CLOS). En aquest cas direm que tenim una *arquitectura metacircular*.



Reflexió estructural (*structural reflection*): Capacitat del llenguatge de programació de proporcionar una *reificació* completa del *programa* que s'està executant i dels *tipus abstractes de dades*.

Reflexió de comportament (*behavioral reflection*): Capacitat del llenguatge de programació de proporcionar una *reificació* completa de la seva pròpia *semàntica* i *implementació* (processador) i de les dades i implementació del *run-time system*

Malenfant et al., A Tutorial on Behavioral Reflection and its Implementation, 1996

La distinció entre **Reflexió estructural** i **Reflexió de comportament** i entre **Introspecció** i **Intercessió** és *ortogonal*.

La primera distinció determina *quin tipus d'accés* tenim a la representació, la segona distinció determina *què podem fer* amb aquesta representació.