You should know that

roundcrisis.com

Let’s learn that

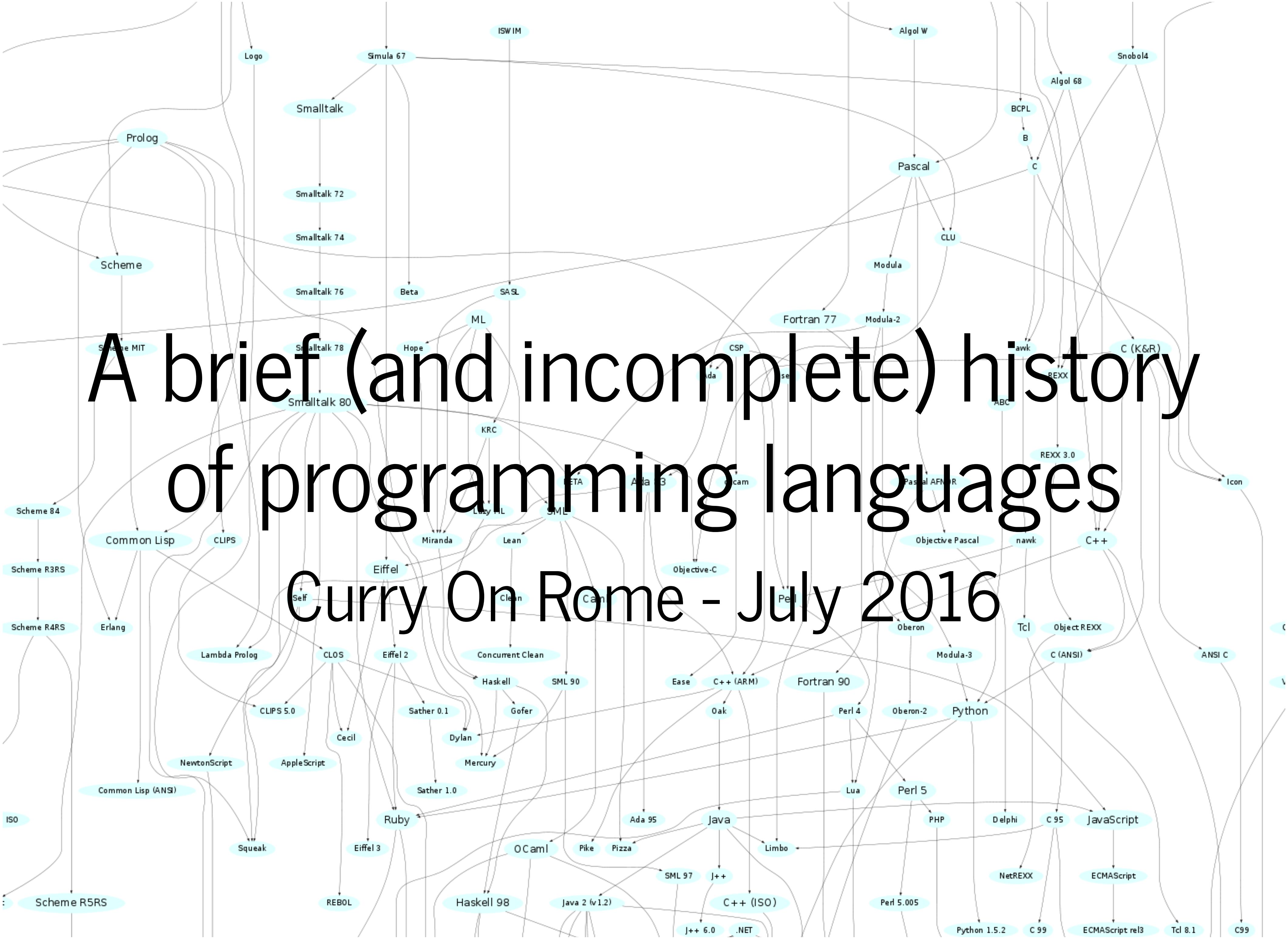
Andrea Magnorsky

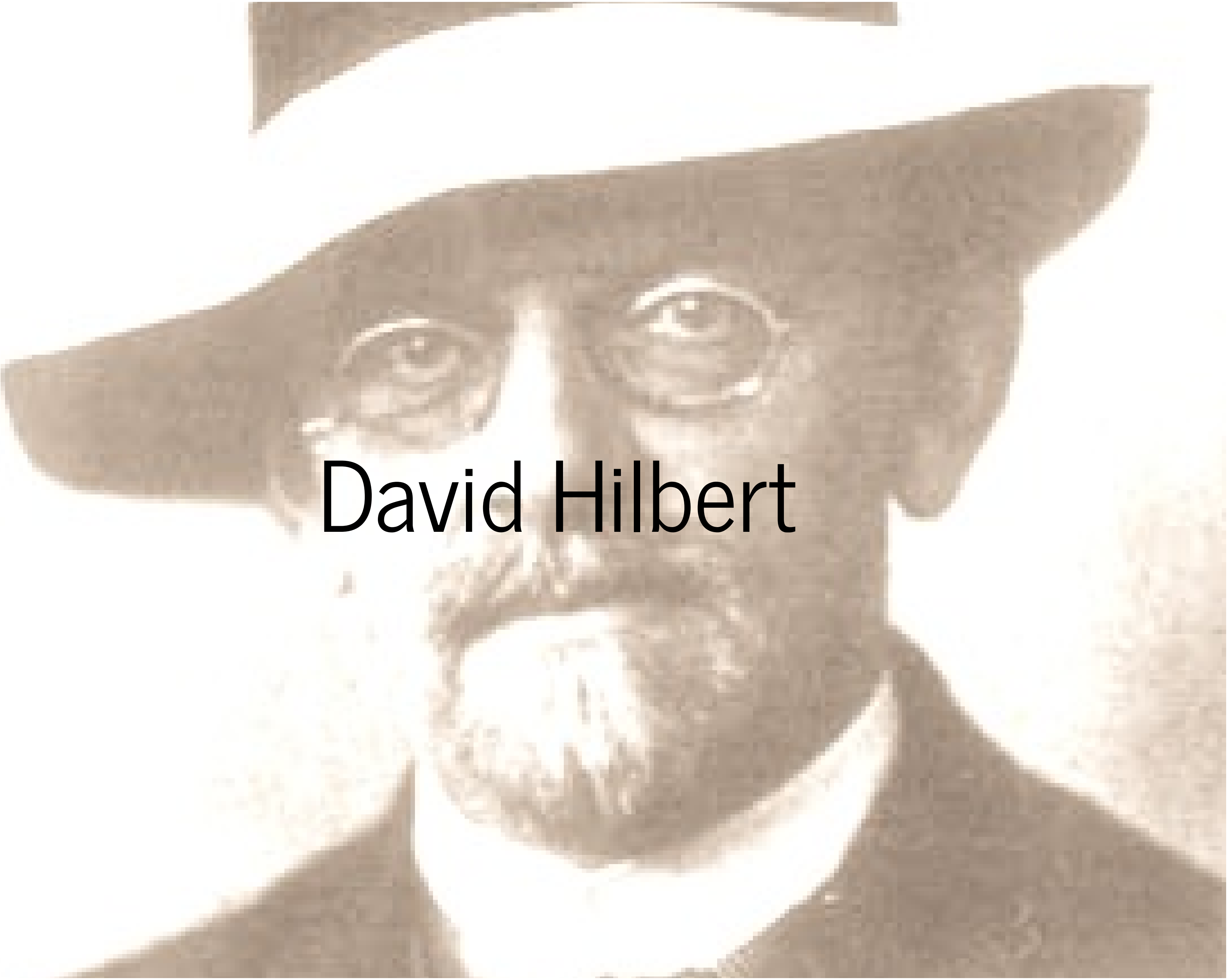
Curry On Rome

Thanks to

A brief (and incomplete) history of programming languages

Curry On Rome - July 2016

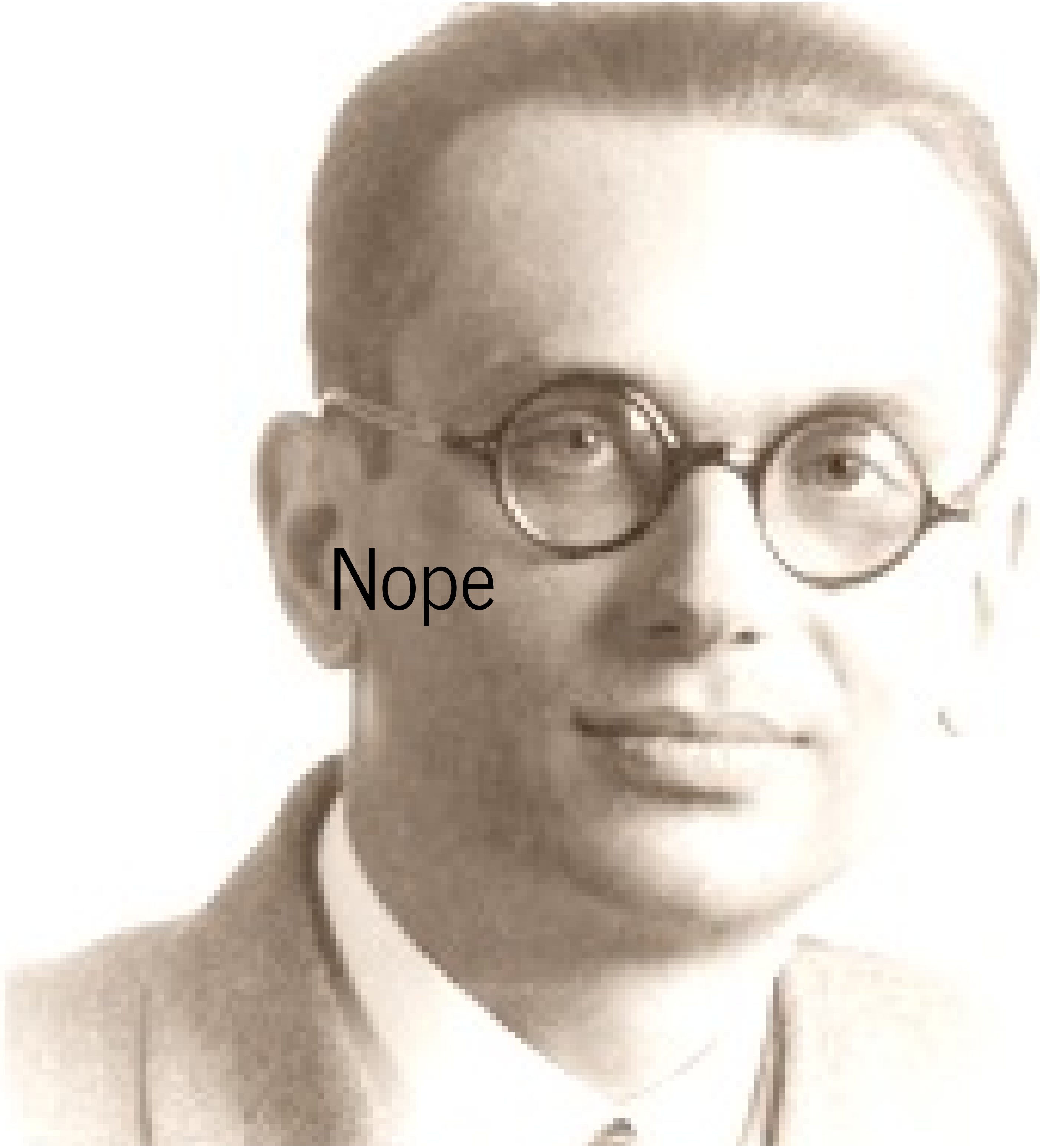



David Hilbert

We must know. We will know.


I HAZ A QUESTION

Can we devise a process to determine in a finite number of operations, whether a first order logic statement is valid?

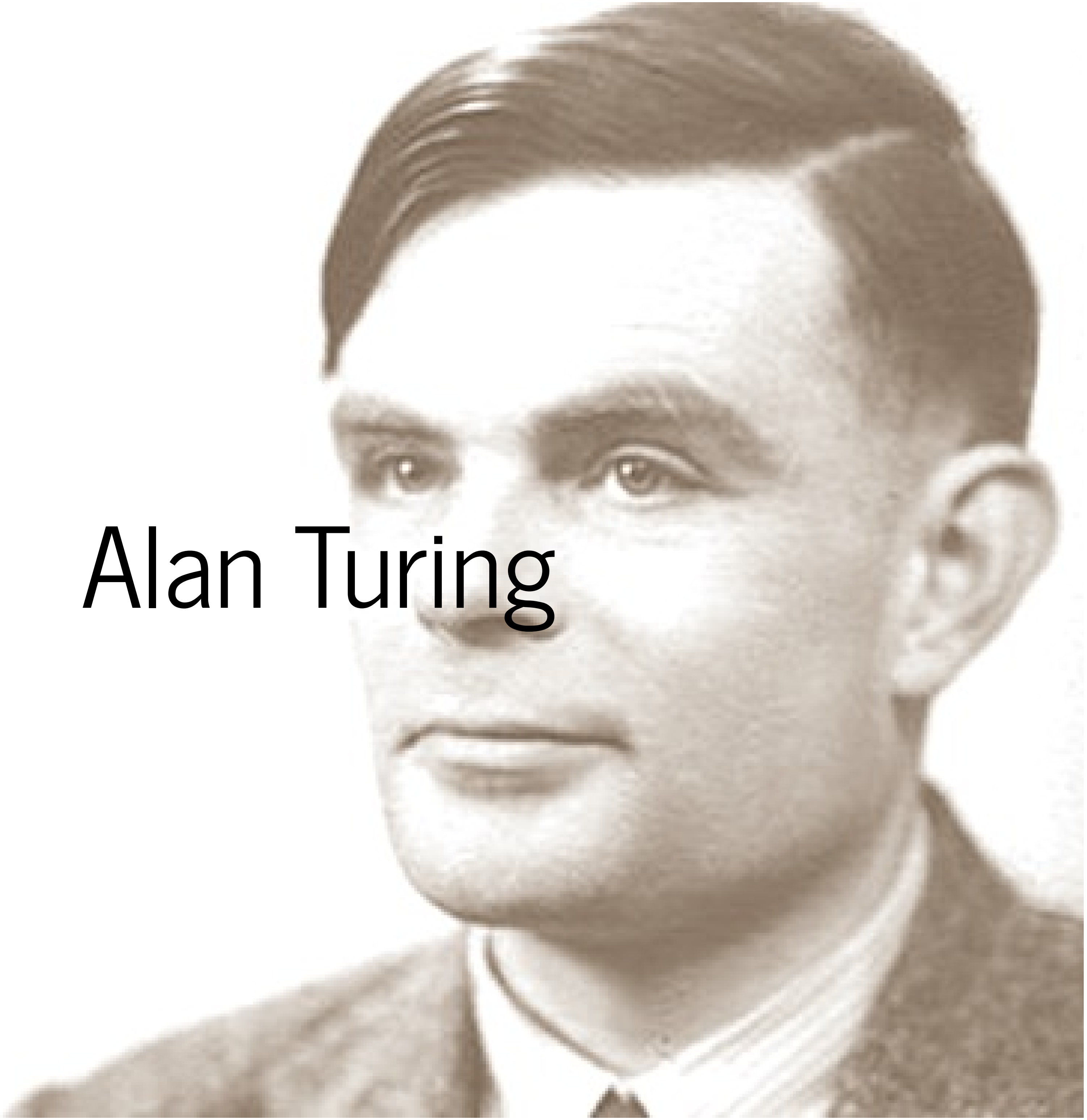

Nope

Kurt Gödel

Alonzo Church

λ Calculus

Alan Turing



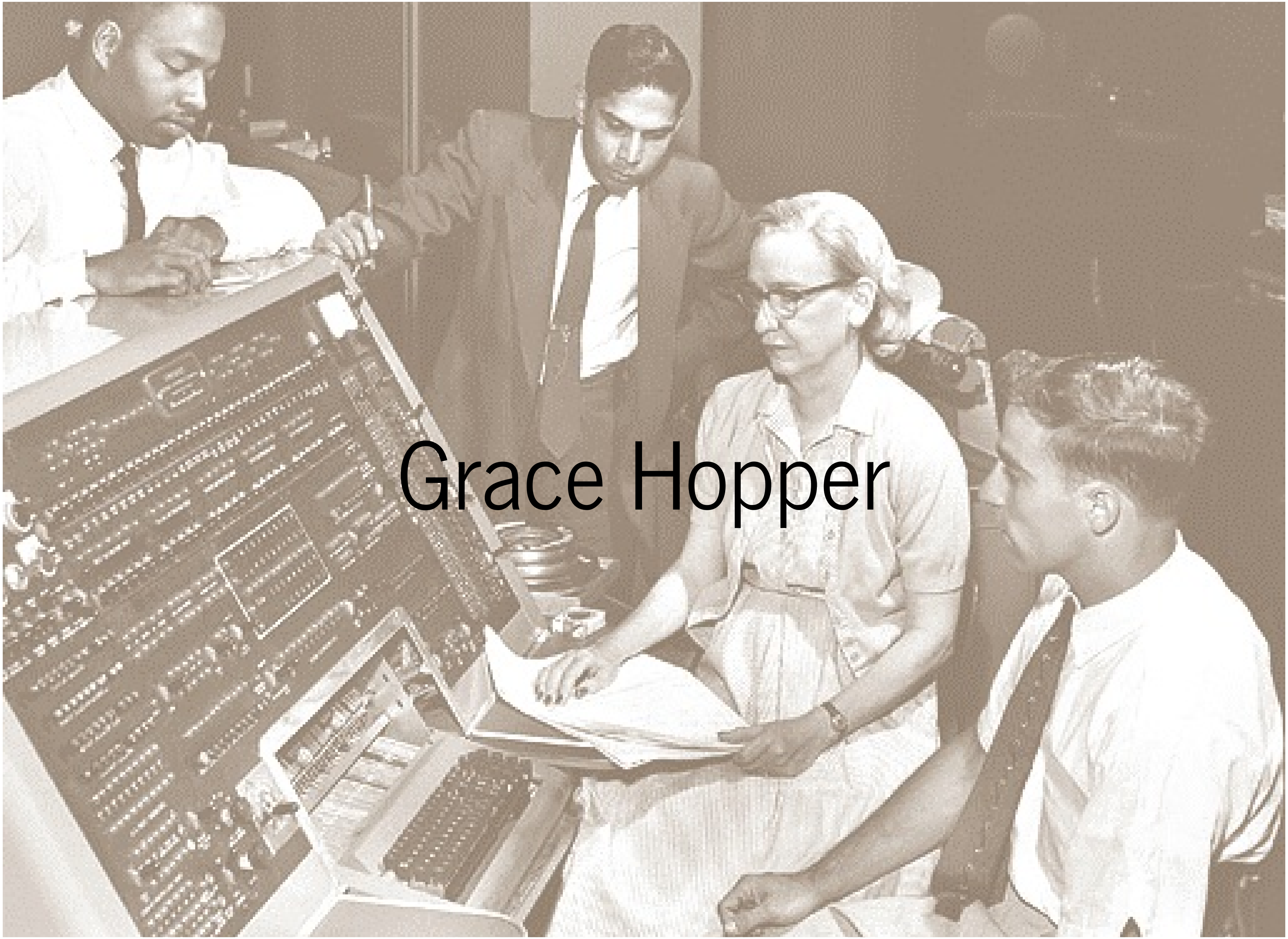
Turing Machine

sauce

Church-Turing Thesis

World war II

Grace Hopper



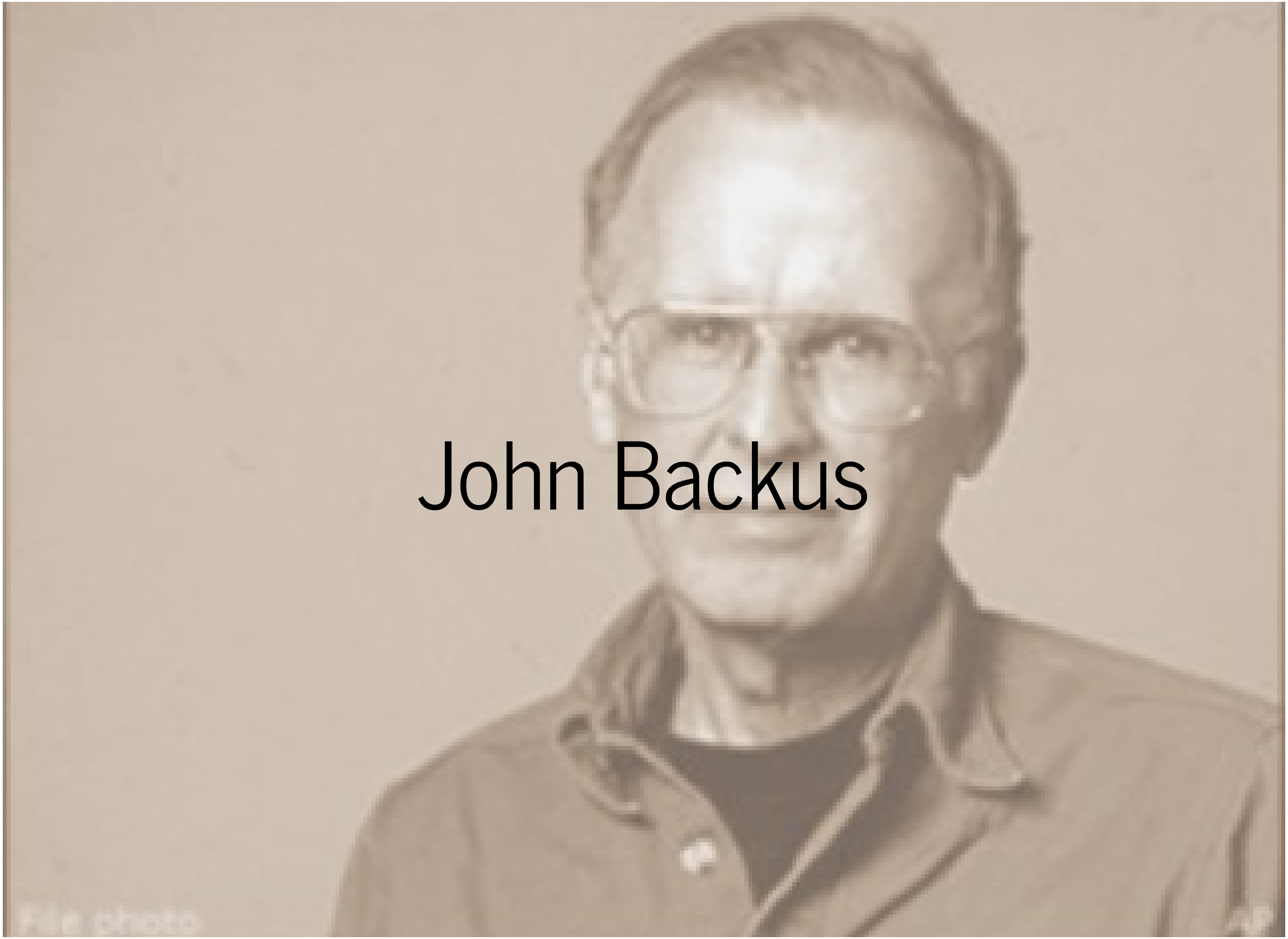
The first compiler: A-0

FLOW-MATIC

```
0) INPUT INVENTORY FILE=A
PRICE FILE=B,
OUTPUT PRICED-INV FILE=C
UNPRICED-INV FILE=D,
HSP D.
1) COMPARE PRODUCT-NO(A) WITH PRODUCT-NO(B)
IF GREATER GO TO OPERATION 10;
IF EQUAL GO TO OPERATION 5;
OTHERWISE GO TO OPERATION 2.
2) TRANSFER A TO D.
3) WRITE ITEM D.
4) JUMP TO OPERATION 8.
5) TRANSFER A TO C.
```

```
 6) MOVE UNIT-PRICE(B) TO UNIT-PRICE(C).
 7) WRITE ITEM C.
 8) READ ITEM A; IF END OF DATA GO TO OPERATION 14.
 9) JUMP TO OPERATION 1.
10) READ ITEM B; IF END OF DATA GO TO OPERATION 12.
11) JUMP TO OPERATION 1.
12) SET OPERATION 9 TO GO TO OPERATION 2.
13) JUMP TO OPERATION 2.
14) TEST PRODUCT-NO(B) AGAINST ZZZZZZZZZZZZ;
    IF EQUAL GO TO OPERATION 16;
    OTHERWISE GO TO OPERATION 15.
15) REWIND B.
16) CLOSE-OUT FILES C, D.
17) STOP. (END)
```

John Backus



Speedcoding
BNF

C ← FOR COMMENT / STATEMENT NUMBER (1–5)	CONTINUATION (6)	FORTRAN STATEMENT (7–72)	IDENTIFICATION (73–80)
C		PROGRAM FOR FINDING THE LARGEST VALUE	
C	X	ATTAINED BY A SET OF NUMBERS	
		DIMENSION A(999)	
		FREQUENCY 30(2,1,10), 5(100)	
		READ 1, N, (A(I), I = 1,N)	
1		FORMAT (I3/(12F6.2))	
		BIGA = A(1)	
5		DO 20 I = 2,N	
30		IF (BIGA-A(I)) 10,20,20	
10		BIGA = A(I)	
20		CONTINUE	
		PRINT 2, N, BIGA	
2		FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)	
		STOP 77777	


John McCarthy

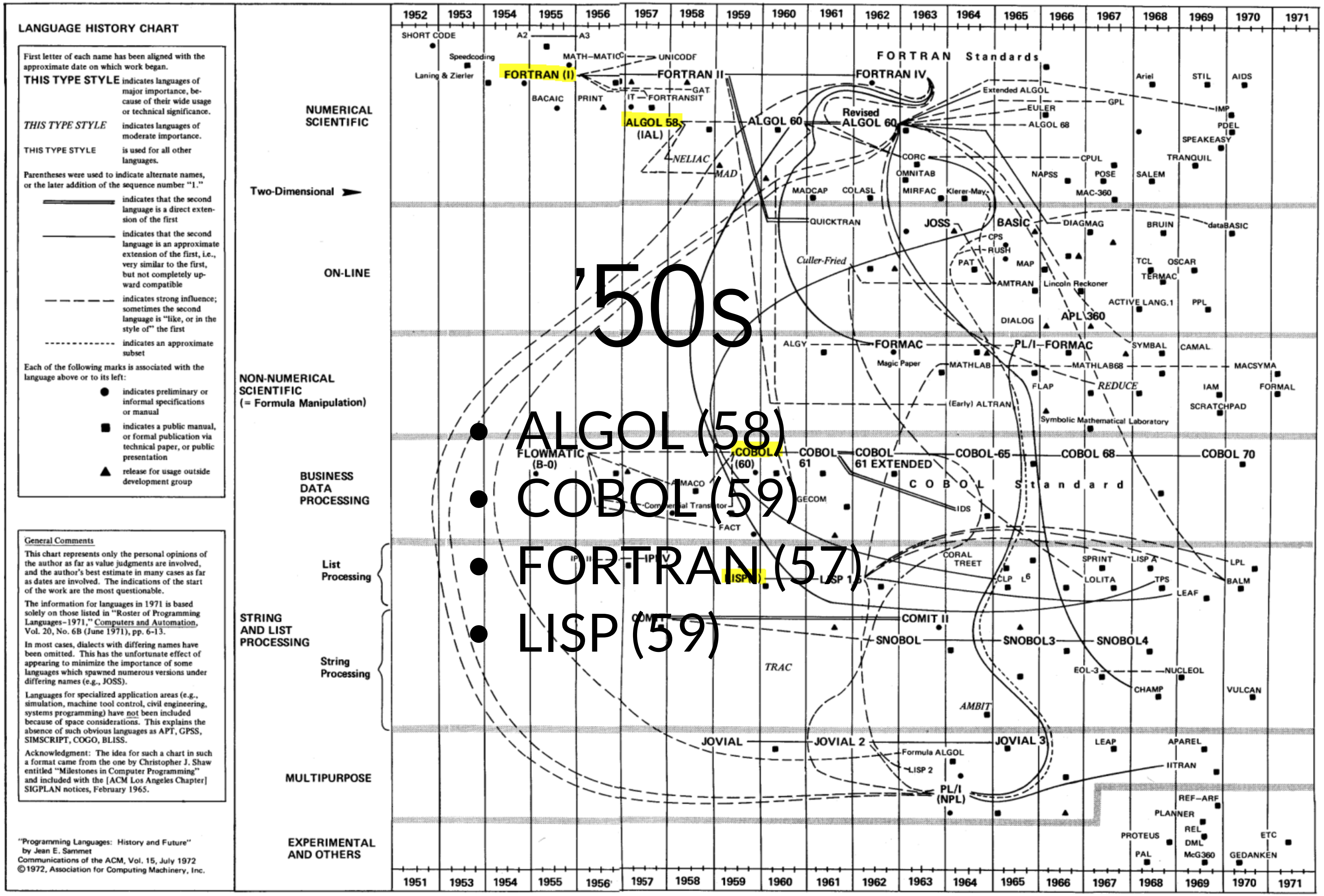
Lisp

AI, time-sharing

ALGOL

’50s

- ALGOL (58)
- COBOL (59)
- FORTRAN (57)
- LISP (59)



LISP

```lisp
(defun is-prime (n)
  (cond ((= 2 n) t)
        ((= 3 n) t)
        ((evenp n) nil)
        (t
           (loop for i from 3 to (isqrt n) by 2
                 never (zerop (mod n i))))))
```

’60s

- APL (62)
- BASIC (64)
- LOGO (67)
- Pascal (69)

APL

$$(\sim T \in T\circ.\times T)/T \leftarrow 1 \downarrow \iota R$$

’70s

- Smalltalk (72)
- ML (73)
- Prolog (72)
- C (72)

Prolog

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y) :- parent_child(Z, X), parent_child(Z,

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

’80s

- Erlang (86)
- SQL (83)
- Miranda (85)
- C++ (83)
- Perl (87)

Erlang

```
-module(mymath).
-export([square/1,fib/1]).

square(Value) -> Value*Value.

fib(0) -> 0;
fib(1) -> 1;
fib(N) when N>1 -> fib(N-1) + fib(N-2).
```

’90s

- Haskell (90)
- Ruby (95)
- Python(91)
- Delphi (95)
- Java (95)
- Visual Basic (91)
- Javascript (95)

Javascript

```
function factorial(n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

’00s

- C# (00)
- Scala (04)
- F# (05)
- Clojure (07)
- D (01)
- Go(07)

D

```
void Quack(Animal)(Animal a)
    if( __traits(compiles, a.Quack()))
{
    a.Quack();
}

struct Duck { void Quack(){ "Quack".writeln; }}

int main(string[] argv) {
    Duck d;
    Quack(d); // good
    Quack(5); // compile time error
}
```

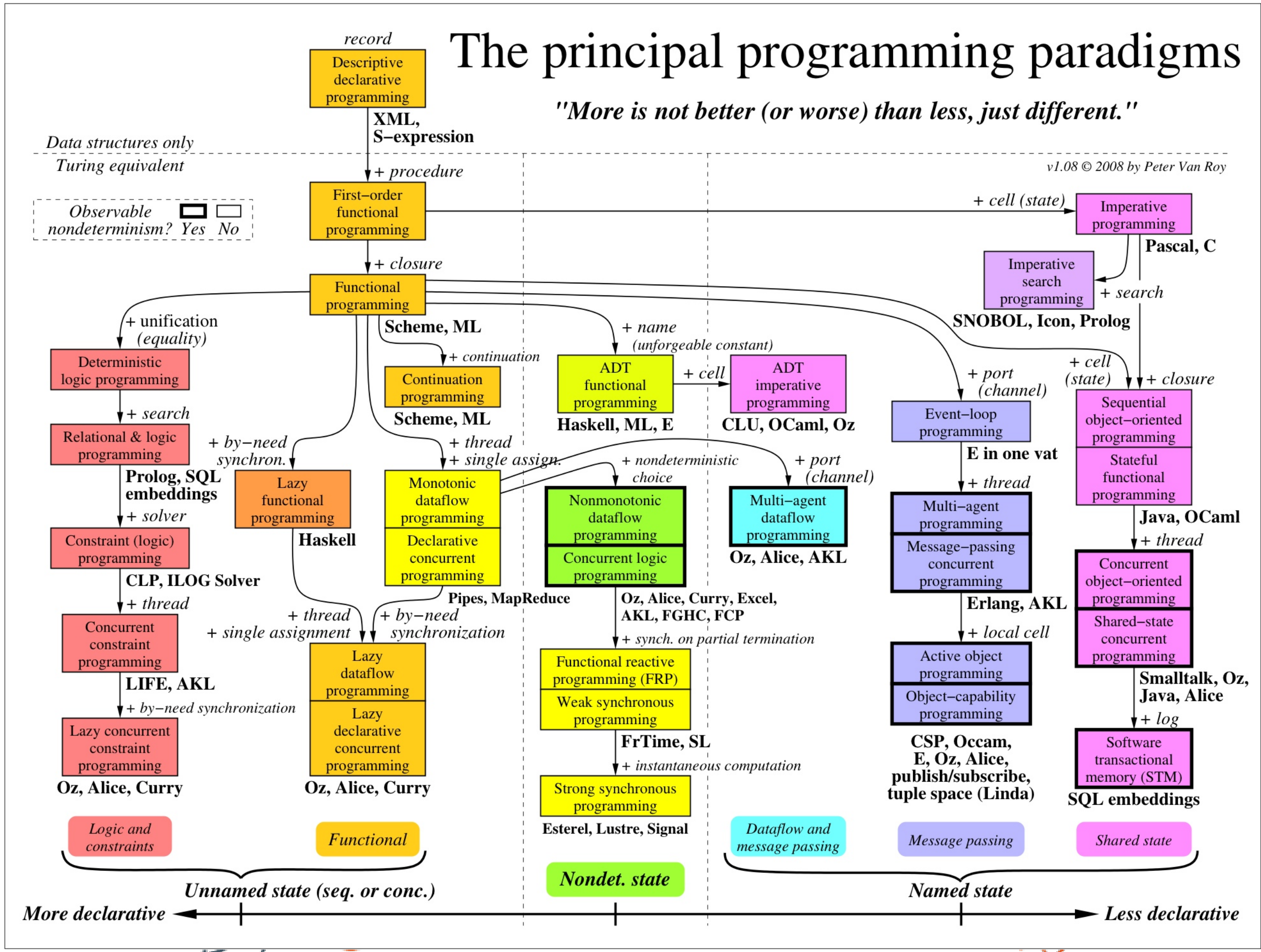
’10s

- Elixir (12)
- Elm (12)
- Rust (10)
- Pony (14)
- Idris (12)

Idris

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect 0 a
  (::) : (x : a) -> (xs : Vect n a) -> Vect (n + 1) a

total
append : Vect n a -> Vect m a -> Vect (n + m) a
append Nil       ys = ys
append (x :: xs) ys = x :: append xs ys
```



Explanations

See "Concepts, Techniques, and Models of Computer Programming".

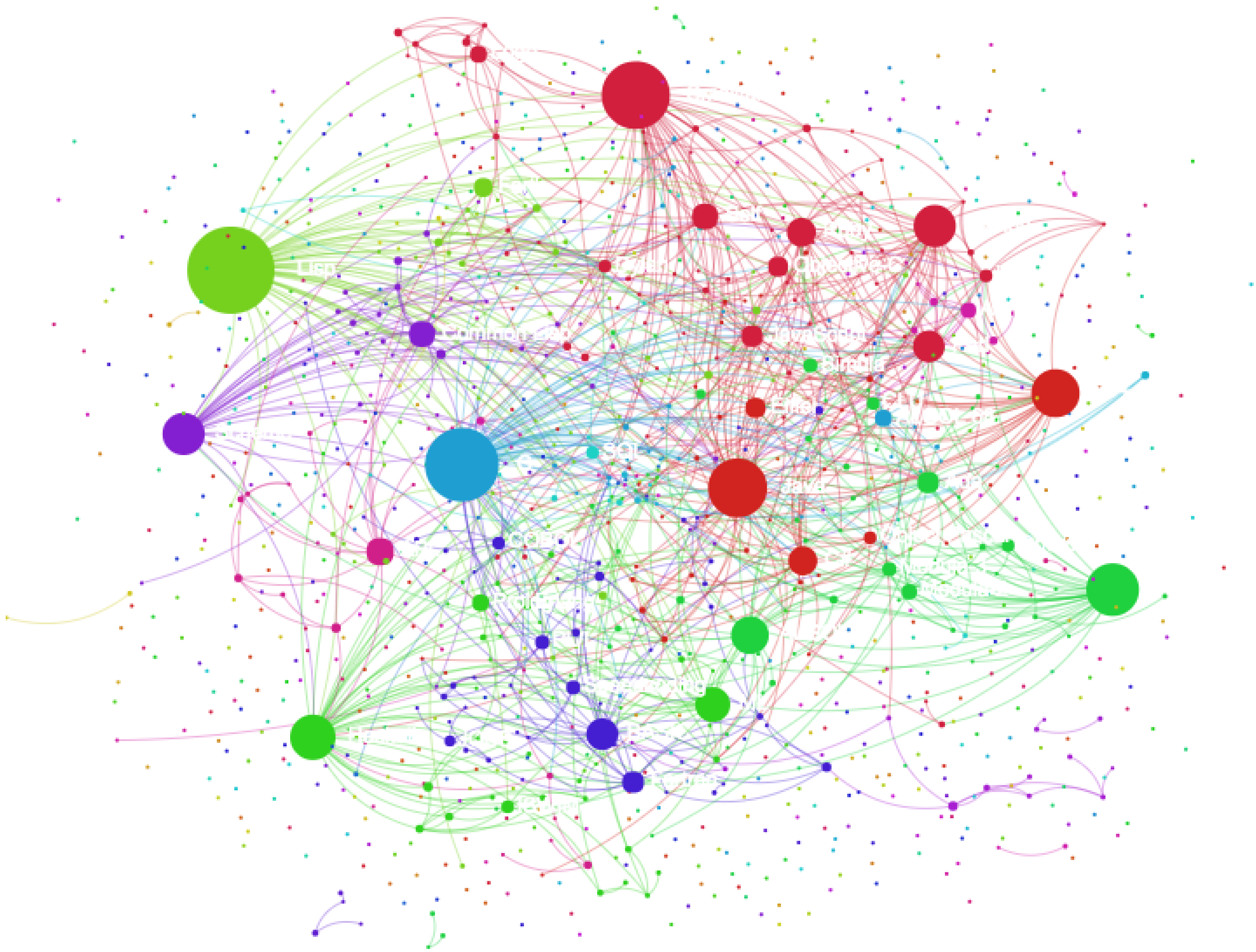
The chart classifies programming paradigms according to their kernel languages (the small core language in which all the paradigm's abstractions can be defined). Kernel languages are ordered according to the creative extension principle: a new concept is added when it cannot be encoded with only local transformations. Two languages that implement the same paradigm can nevertheless have very different "flavors" for the programmer, because they make different choices about what programming techniques and styles to facilitate.

When a language is mentioned under a paradigm, it means that part of the language is intended (by its designers) to support the paradigm without interference from other paradigms. It does not mean that there is a perfect fit between the language and the paradigm. It is not enough that libraries have been written in the language to support the paradigm. The language's kernel language should support the paradigm. When there is a family of related languages, usually only one member of the family is mentioned to avoid clutter. The absence of a language does not imply any kind of value judgment.

State is the ability to remember information, or more precisely, to store a sequence of values in time. Its expressive power is strongly influenced by the paradigm that contains it. We distinguish four levels of expressiveness, which differ in whether the state is unnamed or named, deterministic or nondeterministic, and sequential or concurrent. The least expressive is functional programming (threaded state, e.g., DCGs and monads: unnamed, deterministic, and sequential). Adding concurrency gives declarative concurrent programming (e.g., synchrocells: unnamed, deterministic, and concurrent). Adding nondeterministic choice gives concurrent logic programming (which uses stream mergers: unnamed, nondeterministic, and concurrent). Adding ports or cells, respectively, gives message passing or shared state (both are named, nondeterministic, and concurrent). Nondeterminism is important for real−world interaction (e.g., client/server). Named state is important for modularity.

Axes orthogonal to this chart are typing, aspects, and domain−specificity. Typing is not completely orthogonal: it has some effect on expressiveness. Aspects should be completely orthogonal, since they are part of a program's specification. A domain−specific language should be definable in any paradigm (except when the domain needs a particular concept).

Metaprogramming is another way to increase the expressiveness of a language. The term covers many different approaches, from higher−order programming, syntactic extensibility (e.g., macros), to higher−order programming combined with syntactic support (e.g., meta−object protocols and generics), to full−fledged tinkering with the kernel language (introspection and reflection). Syntactic extensibility and kernel language tinkering in particular are orthogonal to this chart. Some languages, such as Scheme, are flexible enough to implement many paradigms in almost native fashion. This flexibility is not shown in the chart.



We must know. We Will know

Thanks :D


Can you hangout? There's a cute guy here that I want you to meet.
No it's my cats birthday
What?

- @SilverSpoon
- roundcrisis.com

A non exhaustive list of the Resources

- Programming languages: History and future (1972 Jean E. Sammet)
- Definition of Turing Machines - Standford Encyclopedia of Philosophy
- This has happened before and will happen again - Strange Loop conference recording- Video
- David Hilbert
- Alan Kay: Computer Applications: A Dynamic Medium for Creative Thought 1972
- The APL Programming Language Source Code
- Roots of computer languages through the ages
- Principal programming paradigms

- Some History of Functional Programming Languages - D. A. Turner
- Visualizing influence relations of programming languages
- Freebase programming language collection
- Turing on computable numbers
- A Programming Language

Photo credits

- history main starting the talk
- "Alonzo Church" by Princeton University. Licensed under Fair use via Wikipedia