The Soul of Computer Science

Salvador Lucas

DSIC, Universitat Politècnica de València (UPV)

Talk at the Universidad Complutense de Madrid

2

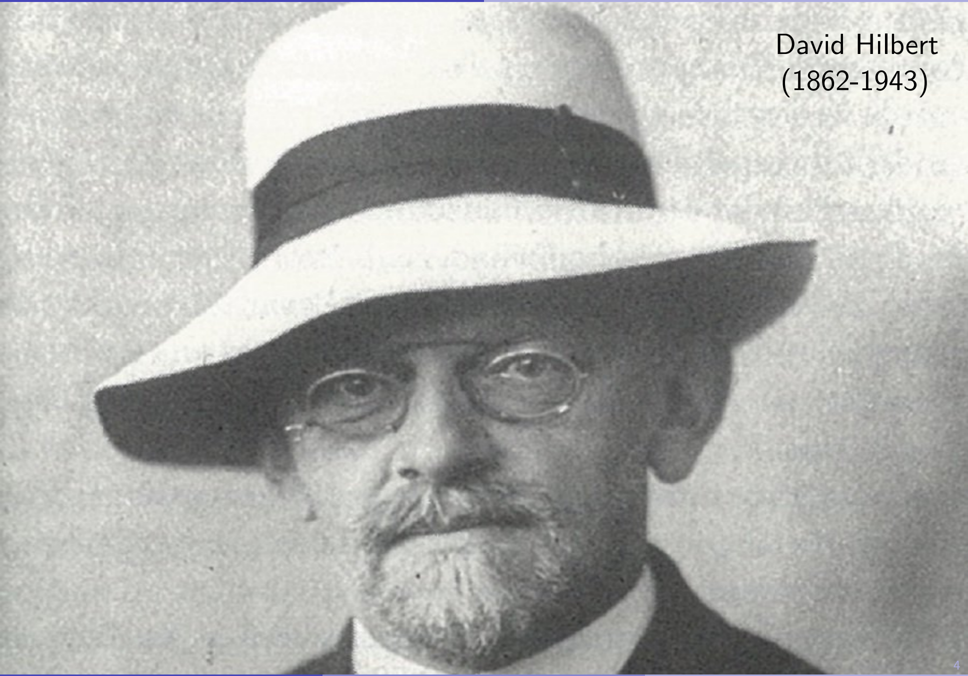
Waves of *Logic* in the history of Computer Science (incomplete list):

0. Hilbert posses "the main problem of mathematical logic" (20's)
1. Church and Turing's logical devices as *effective methods* (1936)
2. Shannon's encoding of *Boolean functions* as circuits (1938)
3. von Neumann's *logical design of an electronic computer* (1946)
4. Floyd/Hoare's logical approach to *program verification* (1967-69)
5. Kowalski's *predicate logic as programming language* (1974)
6. Hoare's challenge of a *verifying* compiler (2003)
7. Berners-Lee's *semantic web* challenge (2006)

Soul

Distinguishing mark of living things (...) responsible for planning and practical thinking (Stanford Encyclopedia of Philosophy)

We can say: *Logic is the soul of Computer Science*!



David Hilbert
(1862-1943)

In his "*Mathematical Problems*" address during the 2^{nd} *International Congress of Mathematicians* (Paris, 1900), he proposed the following:

10^{th} Hilbert's problem

Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a *process* according to which it can be determined *by a finite number of operations* whether the equation is *solvable in rational integers*.

A *diophantine equation* is just a polynomial equation $P(x_1, \ldots, x_n) = 0$ where only *integer solutions* are accepted.

In logical form, we ask whether the following sentence is true:

$$(\exists x_1 \in \mathbb{N}, \ldots, \exists x_n \in \mathbb{N})\ P(x_1, \ldots, x_n) = 0$$

In his "*Mathematical Problems*" address during the 2^{nd} *International Congress of Mathematicians* (Paris, 1900), he proposed the following:

10^{th} Hilbert's problem

Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a *process* according to which it can be determined *by a finite number of operations* whether the equation is *solvable in rational integers*.

A *diophantine equation* is just a polynomial equation $P(x_1, \ldots, x_n) = 0$ where only *integer solutions* are accepted.

In logical form, we ask whether the following sentence is true:

$$(\exists x_1 \in \mathbb{N}, \ldots, \exists x_n \in \mathbb{N})\ P(x_1, \ldots, x_n) = 0$$

There is no solution!

In 1970, Yuri Matiyasevich proved it *unsolvable*, i.e., there is no such 'process'. How could Matiyasevich reach such a conclusion?

In his 1917 address "*Axiomatic thought*" before the Swiss Mathematical Society, Hilbert starts a new quest on the *foundations of mathematics*.

Hilbert is concerned with:

1. the problem of the *solvability in principle of every mathematical question*,
2. the problem of the subsequent *checkability* of the results of a mathematical investigation,
3. the question of a *criterion of simplicity* for mathematical proofs,
4. the question of the relationships between *content* and *formalism* in mathematics and logic,
5. and finally the problem of the *decidability* of a mathematical question in a finite number of operations.

Hilbert's formalist approach to mathematics

It is well-known that Hilbert's approach to these questions took *logic* as the main framework to approach these issues

In his 1928 book *Principles of Theoretical Logic* (with W. Ackermann), he writes:

> *one can apply the first-order calculus in particular to the axiomatic treatment of theories...*

His plan is using logic as a *universal calculus* in mathematics, so that:

> *one can expect that a systematic, so-to-say computational treatment of logical formulas is possible, which would somewhat correspond to the theory of equations in algebra.*

In his 1928 book *Principles of Theoretical Logic* (with W. Ackermann), he writes:

> *one can apply the first-order calculus in particular to the axiomatic treatment of theories...*

His plan is using logic as a *universal calculus* in mathematics, so that:

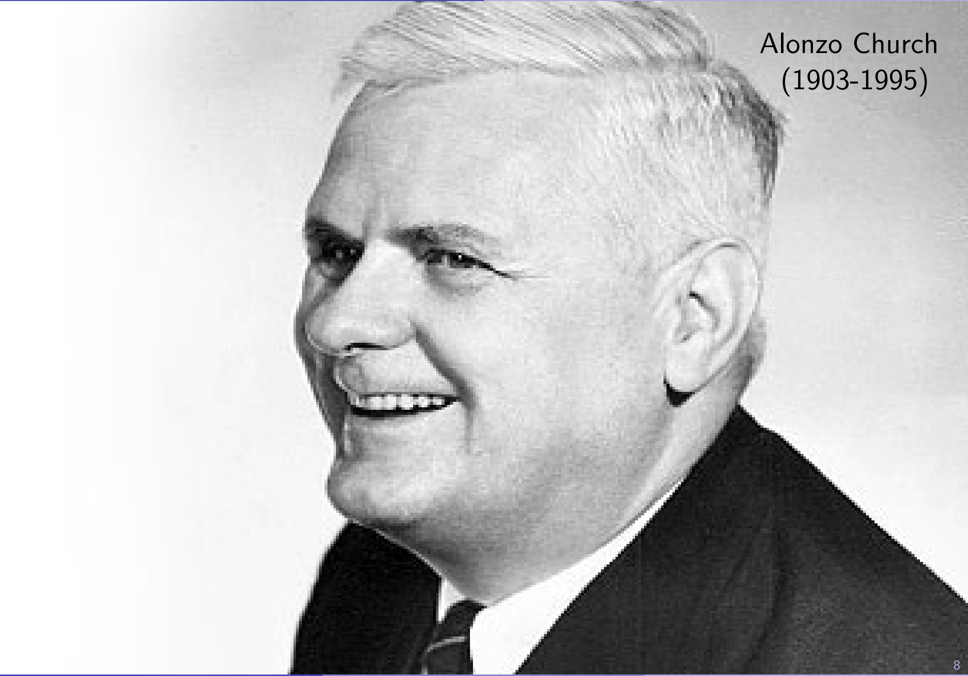
> *one can expect that a systematic, so-to-say computational treatment of logical formulas is possible, which would somewhat correspond to the theory of equations in algebra.*

The Decision Problem

The decision problem is solved if one knows a *process* which, *given a logical expression*, permits the determination of its *validity* resp. *satisfiability*.

For Hilbert, the decision problem is

> *the main problem of mathematical logic ...the discovery of a general decision procedure is still a difficult unsolved problem*


Alonzo Church
(1903-1995)

In his 1936 paper, *An Unsolvable Problem of Elementary Number Theory*, Alonzo Church proposes

> *a definition of effective calculability which is thought to correspond satisfactorily to the somewhat vague intuitive notion in terms of which problems of this class are often stated,*

In his 1936 paper, *An Unsolvable Problem of Elementary Number Theory*, Alonzo Church proposes

> *a definition of effective calculability which is thought to correspond satisfactorily to the somewhat vague intuitive notion in terms of which problems of this class are often stated,*

Church's proposal of an *effective method* was the following formalism, intended to capture the essentials of using functions in mathematics:

Definition (Lambda calculus)

Syntax:

$$M ::= \underbrace{x}_{\textit{variable}} \mid \underbrace{\lambda x.M}_{\textit{abstraction}} \mid \underbrace{M\ N}_{\textit{application}}$$

β-reduction:

$$\underbrace{(\lambda x.M)N}_{\textit{redex}} \to_\beta M[x \mapsto N]$$

Church showed that *arithmetics* can be *encoded* into this calculus.

Then, he *claimed* the following:

Church's Thesis (1936)

Every *effectively calculable* function of positive integers can be *λ-defined*, i.e., defined by means of an expression of the λ-calculus and computed using β-reduction.

Then, the decision problem is considered, in particular, for the elementary number theory. As announced in the introduction, this effort lead

> *to show, by means of an example, that not every problem of this class is solvable.*

Church showed that *arithmetics* can be *encoded* into this calculus.

Then, he *claimed* the following:

Church's Thesis (1936)

Every *effectively calculable* function of positive integers can be *λ-defined*, i.e., defined by means of an expression of the λ-calculus and computed using β-reduction.

Then, the decision problem is considered, in particular, for the elementary number theory. As announced in the introduction, this effort lead

> *to show, by means of an example, that not every problem of this class is solvable.*

The Decision Problem cannot be solved!

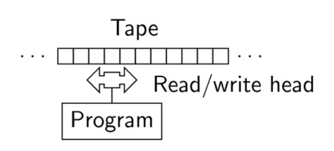
Church showed that, indeed, there are *logical expressions* whose *validity* cannot be established by using his *effective method*.

Under the *assumption* of his *thesis*, no 'process' is able to do the work.


Alan M. Turing
(1912-1954)

Turing machines

In his 1936 paper, *On Computable Numbers, With an Application to the Entscheidungsproblem*, Turing proposes another *computing device*. He called them *a-machines*:



Cells in the tape may be *blank* or contain a *symbol* (e.g., '0' or '1'). The *head* examines only one cell at a time (the *scanned cell*). The machine is able to adopt a number of different *states*. According to this,

1. The head prints a symbol on the scanned cell and *moves* one cell to the *left* or to the *right*.
2. The state changes.

Turing showed how arithmetic computations can be *dealt* with his machine.

In Section 11 of his 1936 paper, he also addresses the *Decision Problem*:

> *"to show that there can be no general process for determining whether a formula is provable"*

and then he rephrases this in terms of his own achievements:

> *"i.e., that there can be no machine which, suplied with any of these formulae, will eventually say whether it is provable."*

When comparing these sentences, it is clear that Turing identifies Hilbert's "general processes" with his own *machine*.

In Section 11 of his 1936 paper, he also addresses the *Decision Problem*:

> *"to show that there can be no general process for determining whether a formula is provable"*

and then he rephrases this in terms of his own achievements:

> *"i.e., that there can be no machine which, suplied with any of these formulae, will eventually say whether it is provable."*

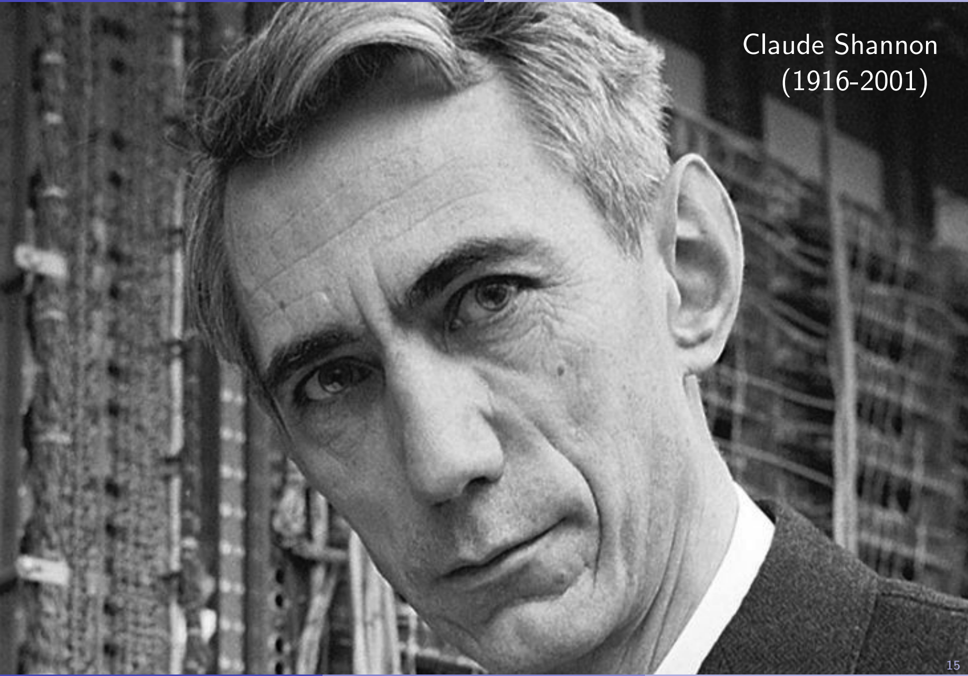
When comparing these sentences, it is clear that Turing identifies Hilbert's "general processes" with his own *machine*.

He also proved that computable functions are λ-definable and vice versa. This leads to the following:

Church-Turing Thesis (1936, )

Every *effectively calculable* function of positive integers is *computable*, i.e., there is a Turing Machine that can be used to obtain its output for a given input.

Turing also describes a *Universal Machine* which can be used to simulate any other (Turing) machine which is then viewed as a *program*.


Claude Shannon
(1916-2001)

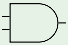
In his 1938 Master Thesis *A Symbolic Analysis of Relay and Switching Circuits*, Claude Shannon showed that *symbolic logic* from George Boole's *Laws of Thought* provides an appropriate mathematical model for the "logic design" of *digital circuits* and *computer components*.

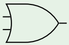
Logic operations and logic gates

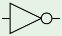
A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	$\neg A$
0	1
1	0







Functions taking boolean inputs and returning boolean values (*Boolean functions*) can be written as a *canonical* combination of $\wedge$, $\vee$, and $\neg$.

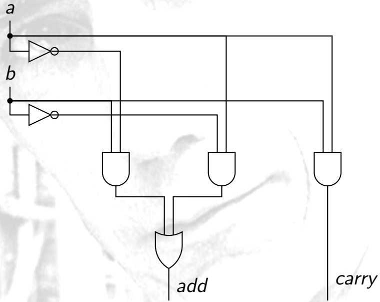
Shannon showed how to obtain a *circuit* to compute such a function

The *addition* of two *bits* a and b can be described by means of *two* truth tables: one for the *addition* and one for any *carry* (to be propagated):

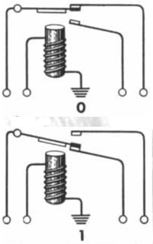
a	b	add
0	0	0
0	1	1
1	0	1
1	1	0

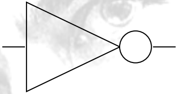
a	b	carry
0	0	0
0	1	0
1	0	0
1	1	1

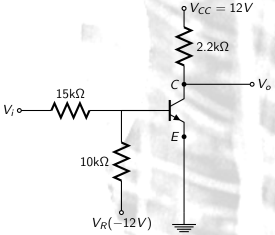
$$add(a,b) = ((\neg a) \wedge b) \vee (a \wedge (\neg b))$$
$$carry(a,b) = a \wedge b$$



Logic gates can be *realized* using different technologies. Shannon considered *relays*, we now use *transistors*:



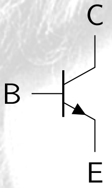




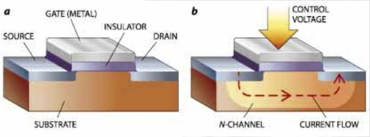
not gate with relays

not gate with transistors

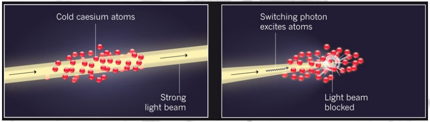
Transistors can also be realized as *electronic* or *molecular* devices:



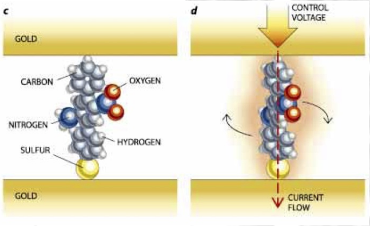
A transistor



Microelectronic



Photonic
(Powell, Nature, June 2013)

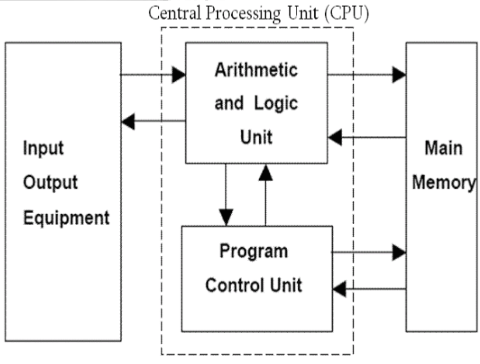


Molecular
(Reed and Tour, Scientific American, June 2000)

The essentials are in the *logical design*. The specific technology is *secondary*!

John von Neumann
(1903-1957)

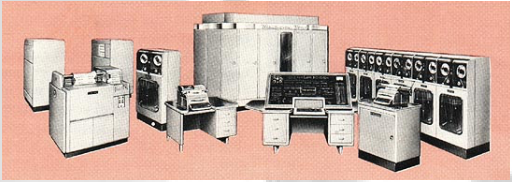
Following his 1945 paper, *First Draft of a Report on the EDVAC*, in a joint paper with Arthur W. Burks and Herman H. Goldstine, John von Neumann proposes a *logical design of an electronic computing instrument*.



- Program and data *stored* in the main memory
- There are *arithmetic*, *memory transfer*, *control*, and *I/O* instructions.
- The control unit *retrieves* and *decodes* instructions
- The arithmetic and logic unit *executes* them

Nothing *substantially new* is added to (Universal) Turing's Machine!

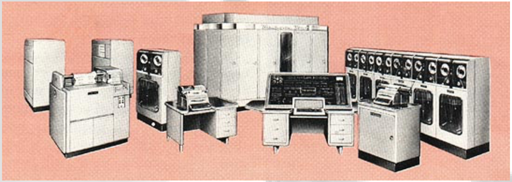
According to Church-Turing's Thesis, other computer architectures (e.g., Harvard's, Parallel, etc.) do not *substantially* improve Turing Machines!



UNIVAC I (1950s)

XXIth Century Supercomputer

According to Church-Turing's Thesis, other computer architectures (e.g., Harvard's, Parallel, etc.) do not *substantially* improve Turing Machines!



UNIVAC I (1950s)

XXIth Century Supercomputer

and *never* will! (!?)

For instance, quoting David Deutsch, prospective computational 'architectures' like *quantum computers*

> *"could, in principle, be built and would have many remarkable properties not reproducing by any Turing machine. These do not include the computation of non-recursive functions..."*

Goldstine and von Neumann also addressed the problem of *planning and coding of problems for an electronic computing instrument*. They wrote:

> *"Coding a problem for the machine would merely be what its name indicates: Translating a meaningful text (the instructions that govern solving the problem under considerations) from one language (the language of mathematics, in which the planner will have conceived the problem, or rather the numerical procedure by which he has decided to solve the problem) into another language (that our code)."*

Goldstine and von Neumann also addressed the problem of *planning and coding of problems for an electronic computing instrument*. They wrote:

> *"Coding a problem for the machine would merely be what its name indicates: Translating a meaningful text (the instructions that govern solving the problem under considerations) from one language (the language of mathematics, in which the planner will have conceived the problem, or rather the numerical procedure by which he has decided to solve the problem) into another language (that our code)."*

However, they soon dismissed this 'simple approach', as they were

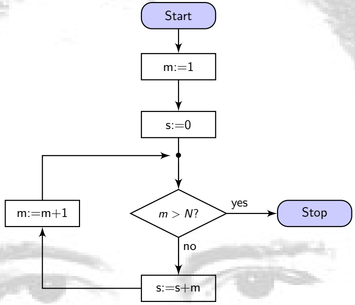
> *"convinced, both on general grounds and from our actual experience with the coding of specific numerical problems, that the main difficulty lies just at this point."*

The main raised point was specifiying the *control* of the execution.

Goldstine and von Neumann introduce *flow diagrams* to *plan the course of the process* and then *extract from this the coded sequence*.

According to this, we proceed as follows:

Add the numbers from 1 to N for some positive N.



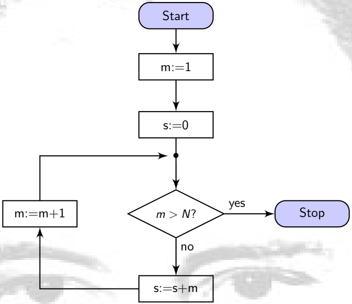

```
integer m s;

s := 0;
m := 1;
while m <= N do
begin
   s := s + m;
   m := m + 1;
end
```

Specification | Control Analysis | Program

According to this, we proceed as follows:

Add the numbers from 1 to N for some positive N.




```
integer m s;

s := 0;
m := 1;
while m <= N do
begin
   s := s + m;
   m := m + 1;
end
```

Specification | Control Analysis | Program

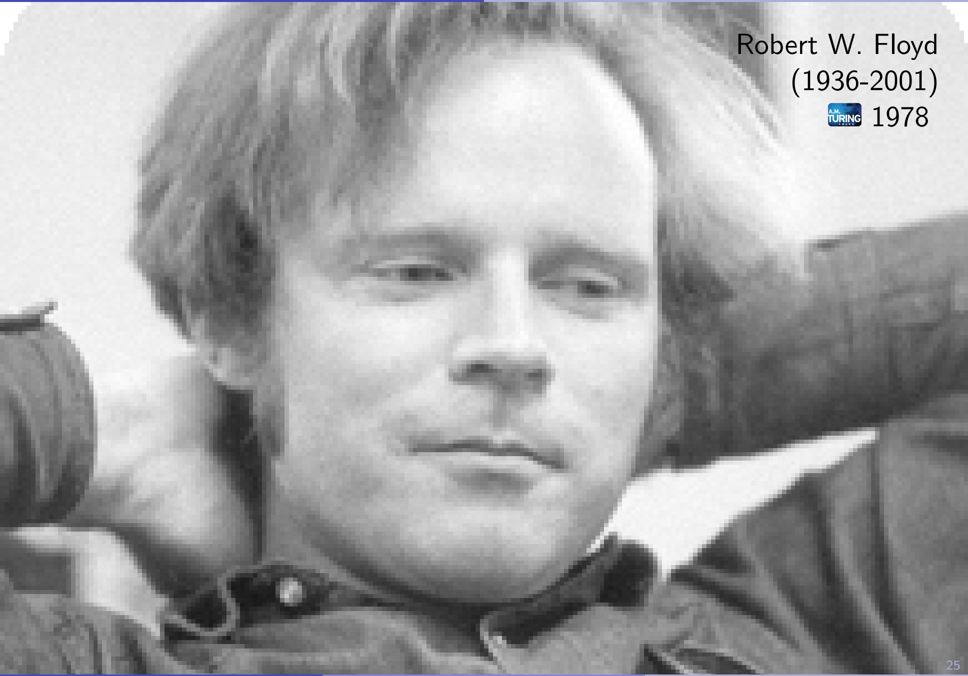
Thus, the following problem arises:

Is the *program* a *solution* to the *specified* problem?

This is a *central problem* in software development. Quoting *Dijkstra*:

...it is *not* our business to make programs; it is our business to design classes of computations that will display a desired behavior.

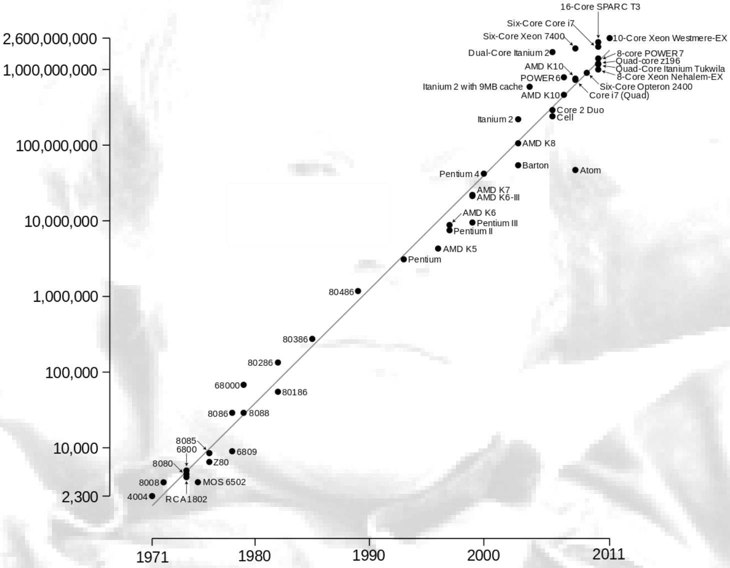
Robert W. Floyd
(1936-2001)
1978



The development of *integrated circuits* in the late fifties led to the *third generation* of computers and to an increase of *speed*, *memory*, and *storage* allowing for *bigger programs* and *concurrency*.

Margaret Hamilton's Apollo XI code (1969)

According to *Moore's law* (*the number of components in integrated circuits doubles every year*), this pile grew up quickly!



In his 1966 paper *Proof of algoritms by general snapshots*, Peter Naur considered the impact of these technological achievements in programming and noticed that

> *"the available programmer competence often is unable to cope with their complexities."*

He made the main steps of program construction explicit as follows:

1. We first have the *description of the desired results in terms of static properties*.
2. We then proceed to construct an algorithm for calculating that result, using *examples* and *intuition* to guide us.
3. Having constructed the algorithm, *we want to prove that it does indeed produce a result having the desired properties*.

In his 1967 landmark paper *Assigning Meanings to Programs*, Robert Floyd pioneered the systematic use of logical expressions to *annotate* flow diagrams so that *properties of programs* could be *logically expressed* and formally *proved*.

In particular, he addressed properties of the form:

> *"If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy relation R_2."*

Floyd's paper was very influential as it showed that "*the specification of proof techniques provides an adequate formal definition of a programming language*" (quoted from Hoare).

Floyd's paper is also celebrated by introducing the first systematic treatment of program *termination proofs* using *well-ordered sets*.

Tony Hoare
born 1934
1980

Hoare's 1969 landmark paper, *An Axiomatic Basis for Computer Programming*, provides a formal calculus to prove program properties.

The calculus concerns the so-called *Hoare's triples* which (today) are written as follows:

$$\{P\}\, S\, \{Q\}$$

where P is a *logical assertion* called the *precondition*, S is the *source program*, and Q is a logical assertion called the *postcondition*.

The interpretation of Hoare's triples is the following:

> *"If the assertion P is true before initation of a program S, then the assertion Q will be true on its completion."*

This provides a way to *specify* software requirements which the *user* wants to see fulfilled by the *program*. The programmer should be able to *guarantee* the *correctness* of the obtained program with respect to such requirements.

$\{N > 0\}$

```
integer m s;

s := 0;
m := 1;
while m <= N do
begin
   s := s + m;
   m := m + 1;
end
```

$\{s = \frac{N(N+1)}{2}\}$

Read as follows: *if the input value N is positive, then, after completing the execution of the program, the output value s contains (according to Gauss' formula) the addition of the numbers from 1 to N, both included*.

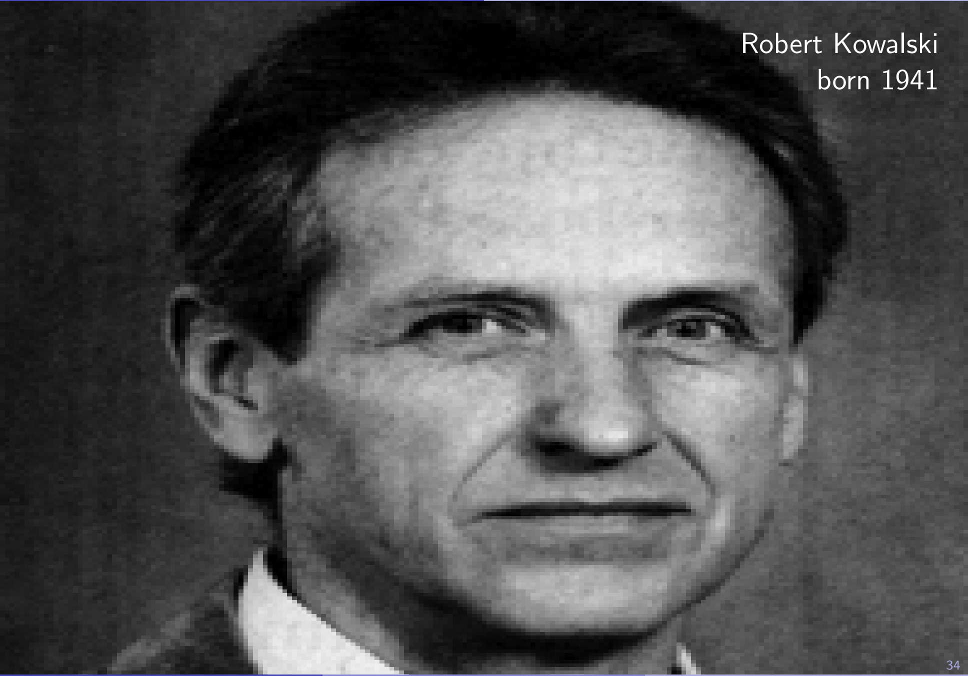
Hoare's calculus provides a way to deal with Hoare's triples so that one can actually *prove* that one such property actually holds.

$$\frac{}{\{P\}\ skip\ \{P\}} \qquad \frac{}{\{P[x \mapsto E]\}\ x := E\ \{P\}}$$

$$\frac{\{P\}\ S\ \{P'\} \qquad \{P'\}\ S'\ \{Q\}}{\{P\}\ S; S'\ \{Q\}} \qquad \frac{\{P \wedge b\}\ S\ \{Q\} \qquad \{P \wedge \neg b\}\ S'\ \{Q\}}{\{P\}\ if\ \ b\ then\ S\ else\ S'\ \{Q\}}$$

$$\frac{\{I \wedge b\}\ S\ \{I\}}{\{I\}\ while\ \ b\ do\ S\ \{I \wedge \neg b\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

$$\frac{\{P\}\ S\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$


Robert Kowalski
born 1941

In the introduction of his 1974 paper *Predicate Logic as Programming Language*, Kowalski writes:

> *"The purpose of programming languages is to enable the communication from man to machine of problems and its general means of solution"*

In the introduction of his 1974 paper *Predicate Logic as Programming Language*, Kowalski writes:

> *"The purpose of programming languages is to enable the communication from man to machine of problems and its general means of solution"*

In contrast to von Neumann, for whom the '*means of solution*' involved the complete description of the *machine control*, Kowalski observes that the following fact:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

could be biased exactly in the opposite way as von Neumann did, so that

"users can restrict their interaction with the computing system to the definition of the logic component, *leaving the determination of the control component to the computer*." (from his 1979 book)

For instance, our running example would be solved by providing a *logical description* of the problem as follows:

```
sum(s(0),s(0))
sum(s(N),S) <= sum(N,R), add(s(N),R,S)
add(0,N,N)
add(s(M),N,s(P)) <= add(M,N,P)
```

where

- terms 0, s(0), ... represent numerals 0, 1, ...
- we read sum(X,Y) as stating that the addition of all numbers from 1 to X is Y.
- we read add(X,Y,Z) as stating that the addition of X and Y is Z.
- we read s(X) as referring to the successor of X.

Each *clause* in the *logic program* above can be interpreted as a (universally quantified) *logical implication* from the predicate calculus.

Kowalski gives a *procedural interpretation* to such clauses.

```
sum(s(0),s(0))
sum(s(N),S) <= sum(N,R), add(s(N),R,S)
add(0,N,N)
add(s(M),N,s(P)) <= add(M,N,P)
```

where

- A *rule* of the form $B \Leftarrow A_1, \ldots, A_n$ is interpreted as a *procedure declaration*. The conclusion B is the *procedure name*. The antecedent $\{A_1, \ldots, A_n\}$ is interpreted as the *procedure body*. It consists of a set of *procedure calls* A_i.
- $B \Leftarrow$ (a rule with an *empty body*) is interpreted as *an assertion of fact* and simply written B.
- $\Leftarrow A_1, \ldots, A_n$ is interpreted as a *goal statement* which asserts the goal of successfully executing all of the procedure calls A_i.

Kowalski gives a *procedural interpretation* to such clauses.

```
sum(s(0),s(0))
sum(s(N),S) <= sum(N,R), add(s(N),R,S)
add(0,N,N)
add(s(M),N,s(P)) <= add(M,N,P)
```

where

- A *rule* of the form $B \Leftarrow A_1, \ldots, A_n$ is interpreted as a *procedure declaration*. The conclusion B is the *procedure name*. The antecedent $\{A_1, \ldots, A_n\}$ is interpreted as the *procedure body*. It consists of a set of *procedure calls* A_i.
- $B \Leftarrow$ (a rule with an *empty body*) is interpreted as *an assertion of fact* and simply written B.
- $\Leftarrow A_1, \ldots, A_n$ is interpreted as a *goal statement* which asserts the goal of successfully executing all of the procedure calls A_i.

In this setting, a *fact* like sum(s(0),s(0)) means that the addition of all numbers from 1 to s(0) *yields* s(0). A computation is a *proof* of this!

Of course, we can also *obtain* the addition from the program!

```
C1: sum(s(0),s(0))
C2: sum(s(N),S) <= sum(N,R), add(s(N),R,S)
C3: add(0,N,N)
C4: add(s(M),N,s(P)) <= add(M,N,P)
```

`sum(s(s(0)),X)` $\{X \mapsto s(s(s(0)))\}$

$\downarrow$ C2, $\{N \mapsto s(0), X \mapsto S\}$

`sum(s(0),R), add(s(s(0)),R,S)` $\{S \mapsto s(s(s(0)))\}$

$\downarrow$ C1, $\{R \mapsto s(0)\}$

`add(s(s(0)),s(0),S)`

$\downarrow$ C4, $\{M' \mapsto s(0), N' \mapsto s(0), S \mapsto s(P')\}$

`add(s(0),s(0),P')` $\{P' \mapsto s(s(0))\}$

$\downarrow$ C4, $\{M'' \mapsto 0, N'' \mapsto s(0), P' \mapsto s(P'')\}$

`add(0,s(0),P'')` $\{P'' \mapsto s(0)\}$

$\downarrow$ C3, $\{N''' \mapsto s(0), P'' \mapsto s(0)\}$

$\square$

The solution is obtained by *propagating* the *blue* bindings (concerning variable `X` in the initial goal) bottom-up:

`X` is bound to `s(s(s(0)))` as expected!

Goldstine and von Neumann's dream:

> *"Coding a problem for the machine would merely be (...) translating (...) the language of mathematics, in which the planner will have conceived the problem (...) into another language (that our code)."*

becomes *feasible*!

Following Kowalski's approach, the *system* in charge of executing the logic program will take care of any *control issues*.

Prolog is the paradigmatic example of a logic programming language.

Correctness for free!?

Since specification and program *coincide*, the program is automatically *correct* without any further proof!

No free lunch!

Although writing and executing '*control-unaware*' programs is possible, in practice it is computationally expensive due to the highly *nondeterministic character* of logic programming computations.

```
sum(s(s(0)),X)
```
$\downarrow$ C2, $\{N \mapsto s(0), X \mapsto S\}$
```
sum(s(0),R), add(s(s(0)),R,S)
```
$\downarrow$ C4, $\{M' \mapsto s(0), N' \mapsto R, S \mapsto s(P')\}$
```
sum(s(0),R), add(s(0),R,P')
```
$\downarrow$ C4, $\{M'' \mapsto 0, N'' \mapsto R, P' \mapsto s(P'')\}$
```
sum(s(0),R), add(0,R,P'')
```
$\downarrow$ C3, $\{N''' \mapsto R, P'' \mapsto R\}$
```
sum(s(0),R)
```
$\downarrow$ C1, $\{R \mapsto s(0)\}$

$\square$

```
C1: sum(s(0),s(0))
C2: sum(s(N),S) <= sum(N,R), add(s(N),R,S)
C3: add(0,N,N)
C4: add(s(M),N,s(P)) <= add(M,N,P)
```

The same solution is obtained but the computation tree is different. And there are other possibilities...

Functional programming relies on Church's *lambda calculus*.

Programs are intended to provide *function definitions* and can be seen as *lambda expressions*.

The execution consists of reducing such expressions. *Haskell* and *ML* are well-known functional languages.

Haskell's version of the running example

```
data Nat = Z | S Nat
sum (S Z) = S Z
sum (S n) = (S n) + sum n
Z     + n = n
(S m) + n = S (m + n)
```

The evaluation of sum (S (S Z)) is *deterministic*:

$$
\begin{array}{rcl}
\mathsf{sum}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})) & \rightarrow & (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})) + \mathsf{sum}\ (\mathsf{S}\ \mathsf{Z}) \rightarrow \mathsf{S}\ ((\mathsf{S}\ \mathsf{Z}) + \mathsf{sum}\ (\mathsf{S}\ \mathsf{Z})) \\
 & \rightarrow & \mathsf{S}\ (\mathsf{S}\ (\mathsf{Z} + \mathsf{sum}\ (\mathsf{S}\ \mathsf{Z}))) \rightarrow \mathsf{S}\ (\mathsf{S}\ (\mathsf{sum}\ (\mathsf{S}\ \mathsf{Z}))) \\
 & \rightarrow & \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))
\end{array}
$$

Correctness for free!

Indeed, there is a Haskell predefined function *sum* that adds the components of a *list* of numbers.

The evaluation of the expression

$$\mathit{sum}\ [1..n]$$

yields exactly what we want.

Here, specification and program *coincide*!

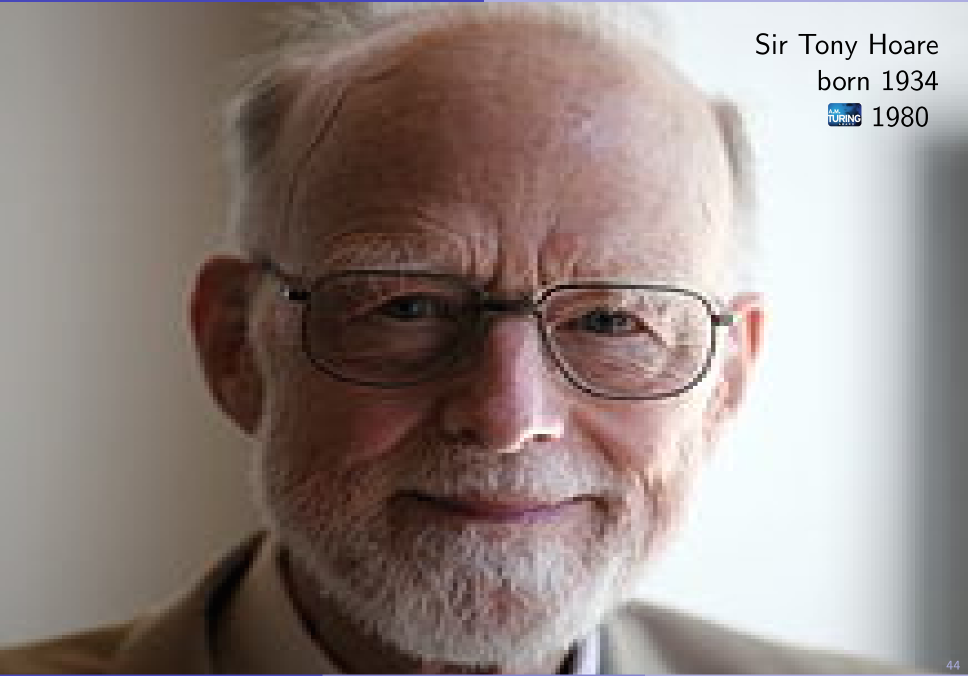
Meseguer's approach to declarative languages as *general logics*

1. *Declarative programs* $\mathcal{S}$ are *theories* of a given *logic* $\mathcal{L}$.
2. *Computations* with $\mathcal{S}$ are implemented as *deductions* in $\mathcal{L}$.
3. *Deductions* proceed according to the *Inference System* $\mathcal{I}$ of $\mathcal{L}$.
4. *Executing* a program $\mathcal{S}$ is proving a *goal* φ using $\mathcal{I}(\mathcal{S})$.

A *logic* $\mathcal{L}$ is often seen as a quadruple $\mathcal{L} = (Th(\mathcal{L}), Form, Sub, \mathcal{I})$, where:

1. $Th(\mathcal{L})$ is the class of *theories* of $\mathcal{L}$,
2. $Form$ is a mapping sending each theory $\mathcal{S} \in Th(\mathcal{L})$ to a set $Form(\mathcal{S})$ of *formulas* of $\mathcal{S}$,
3. Sub is a mapping sending each $\mathcal{S} \in Th(\mathcal{L})$ to its set $Sub(\mathcal{S})$ of *substitutions*, with $Sub(\mathcal{S}) \subseteq [Form(\mathcal{S}) \to Form(\mathcal{S})]$, and
4. $\mathcal{I}$ is a mapping sending each $\mathcal{S} \in Th(\mathcal{L})$ to a subset $\mathcal{I}(\mathcal{S})$ of *inference rules* $\frac{B_1 \ldots B_n}{A}$ for $\mathcal{S}$.

Prolog, *Haskell*, and *ML* can be seen as examples of this approach. Other examples are *CafeOBJ*, *OBJ*, *Maude*, etc.



Sir Tony Hoare
born 1934
1980

In 1996, a tiny error in a part of the flight control software of the Ariane V rocket led to the following:

In 1996, a tiny error in a part of the flight control software of the Ariane V rocket led to the following:

The component had been frequently *tested* on *previous* Ariane IV flights...

Yesterday!: `http://www.nature.com/news/computing-glitch-may-have-doomed-mars-lander-1.20861`

The most likely culprit is a flaw in the crafts software or a problem in merging the data coming from different sensors, which may have led the craft to believe it was lower in altitude than it really was, says Andrea Accomazzo, ESAs head of solar and planetary missions.

In his 2003 paper *The Verifying Compiler: A Grand Challenge for Computing Research*, Tony Hoare proposed

> *"the construction of a verifying compiler that uses mathematical and logical reasoning to check the correctness of the programs that it compiles".*

The compiler is not expected to 'work alone' but

> *"in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components".*

Programmers would specify correctness criteria by means of

> *"types, assertions, and other redundant annotations associated with the code of the program."*

Some progress has been made in this project. A number of *tools* as the ones demanded by Hoare have been developed so far.

Microsoft's verification tool *Dafny*: `http://rise4fun.com/Dafny/`

dafny Microsoft Research

Is this program correct?

```
function gauss(i:int):int
{
  i*(i+1)/2
}

method sum(n: int) returns (s: int)
  requires 0 < n
  ensures s == gauss(n)
{
  s := 0;
  var m := 1;
    while m <= n
    invariant m<=n+1
    invariant s == gauss(m-1)
    {
      s := s + m;
      m := m + 1;
    }
}
```

▶ tutorial home video permalink

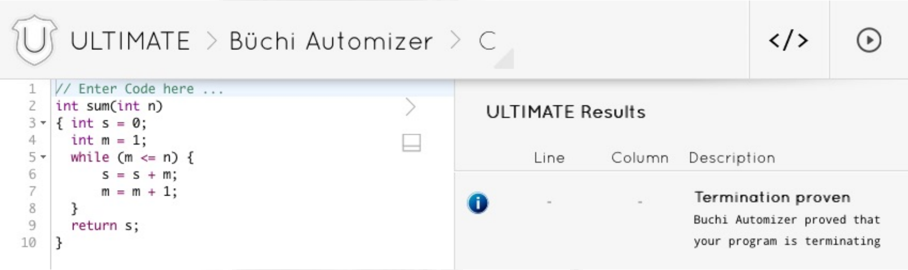
'▶' shortcut: Alt+B

```
Dafny program verifier finished with 3 verified, 0 errors
```

- The user *provides* the *preconditions* (*requires*) and *postconditions* (*ensures*).
- The user can be *asked* to provide some assertions, like loop *invariants*.
- Full automation is possible but difficult (in particular, not possible for this program example).

Ultimate termination tool:

https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/



A completely *automatic* proof is possible in this case!

This *magic* is possible due to the use of

- Propositional satisfiability checking techniques (SAT)
- Decidable logics (FOL with *unary predicates*, *Presburger's* arithmetic, FOL of the *Real Closed Fields*, etc.)
- Techniques for checking propositional satisfiability *modulo theories* (SMT) techniques
- Constraint solving
- Abstract interpretation
- Theorem proving *tools* (HOL, ACL2, Coq, ...)
- Model checking
- ...

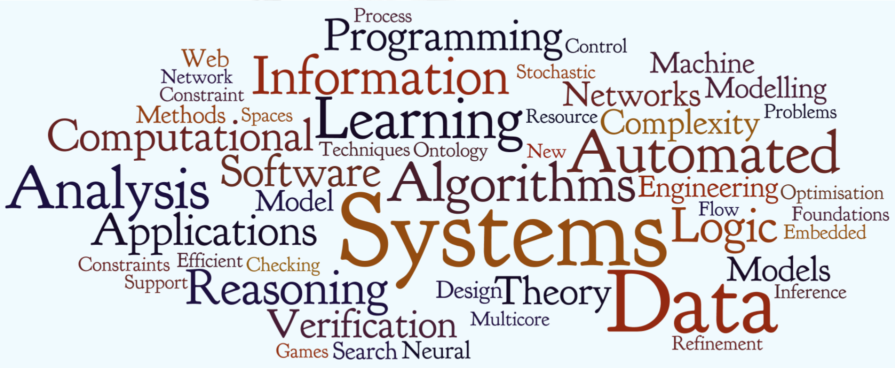
Most of these techniques are not really new, but they have been recently *implemented*, *combined*, and *improved* in different ways so that we can now use them in practice!

Sir Tim Berners-Lee
born 1955

Tim Berners-Lee launched the *first* web site by August 1991

In 2006 he reported the existence of about *10 billion pages* on the now called *World Wide Web*

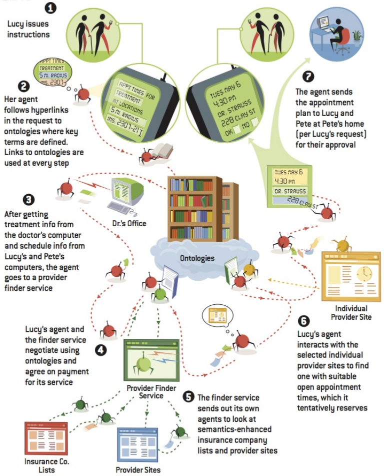
Search engines can be used to uncover themes embodied in such documents and retrieve them to prospective *readers*:



This is quite a lot, but is it *all*?

In his talk during the first WWW conference (1994), he said the following:

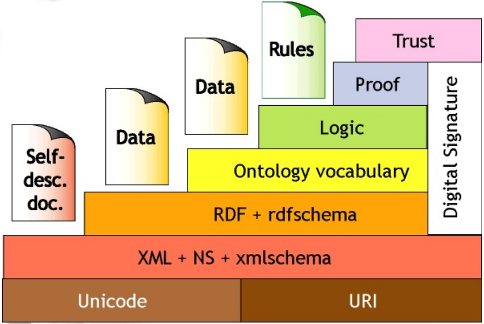
> *The web is a set of nodes and links. To a user this has become an exciting world, but there is very little machine-readable information there... To a computer is devoid of meaning.*



From Berners-Lee, Cailliau, and Lassila's Scientific American paper (May 2001)

Then, Berners-Lee proposes the following:

> *Adding semantics to the web involves two things: allowing documents which have information in machine-readable forms, and allowing links to be created with relationship values.*



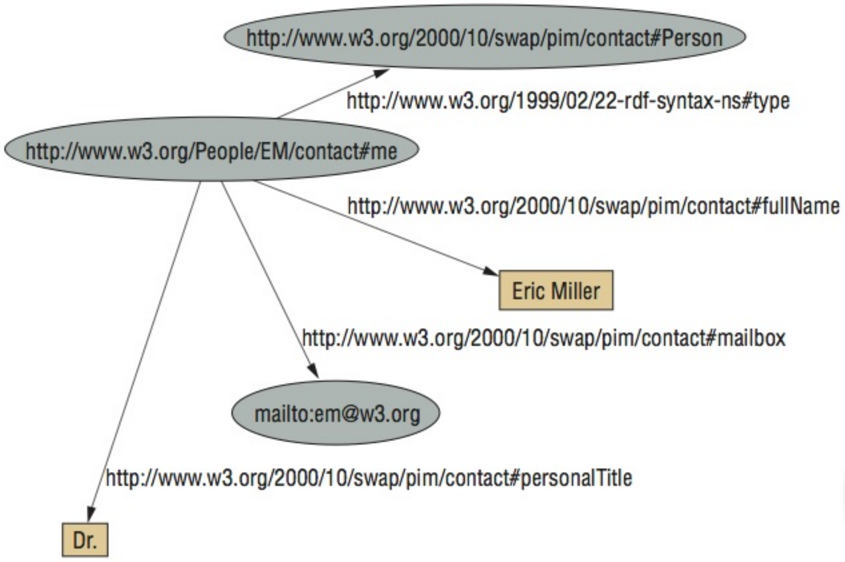
The main ingredients

The *Resource Description Framework* (RDF): a scheme for defining information on the Web. *Ontologies*: Collections of RDF statements

RDF is a *description logic* which can be seen as a *restriction* of first-order logic that improves on the *complexity* and *decidability* problems of FOL.

RDF syntax		FOL syntax	
Name	Concept	Correspondent	Name
Triple	$\text{Subject} \xrightarrow{\text{Predicate}} \text{Object}$	p(s,o)	*Atom*
Graph	Set of triples	Conjunction of atoms	*Theory*

- Nodes in *triples* and in the *graph* are:
  1. *Internationalized Resource Identifiers* (IRIs) or *literals*, which denote *resources* (documents, physical things, abstract concepts, numbers,...)
  2. *Blank nodes* (think of them as existentially quantified *variables*)
- Arcs are labelled by a *predicate*, which is also an IRI and denotes a *property*, i.e., a resource that can be thought of as a binary relation.



Towards the *Semantic Web* !

Semantics of RDF is given as follows (compare with *First-Order Logic*):

RDF semantics		FOL semantics	
Name	Symbol	Correspondent	Name
Resources	IR	A	*Domain*
Properties	IP	–	–
Extension	$IEXT \in IP \to \mathcal{P}(IR \times IR)$	$R \subseteq A \times A$	*Relation*
IRI interp.	$IS \in IRI \to (IR \cup IP)$	$\mathcal{I}$	*Interpretation*
Literal int.	$IL \in Literals \to IR$	$\mathcal{I}$	*Interpretation*
Blank int.	$A \in Blank \to IR$	α	*Var. valuation*

Define a mapping $[I + A]$ to be I on *IRIs* and *literals* and A on *blank nodes*. RDF graphs are given *truth values* as follows:

- If E is a ground triple $\langle s, p, o\rangle$, then $I(E) = true$ if $I(p) \in IP$ and $(I(s), I(o)) \in IEXT(I(p))$; otherwise, $I(E) = false$.
- If E is a triple containing a *blank* node, then $I(E) = true$ if $[I + A](E) = true$ for some $A \in Blank \to IR$; otherwise, $I(E) = false$..
- If E is a graph, then $I(E) = true$ if $[I + A](E) = true$ for some $A \in Blank \to IR$; otherwise, $I(E) = false$..

Semantics of RDF is given as follows (compare with *First-Order Logic*):

RDF semantics		FOL semantics	
Name	Symbol	Correspondent	Name
Resources	IR	A	*Domain*
Properties	IP	–	–
Extension	$IEXT \in IP \to \mathcal{P}(IR \times IR)$	$R \subseteq A \times A$	*Relation*
IRI interp.	$IS \in IRI \to (IR \cup IP)$	$\mathcal{I}$	*Interpretation*
Literal int.	$IL \in Literals \to IR$	$\mathcal{I}$	*Interpretation*
Blank int.	$A \in Blank \to IR$	α	*Var. valuation*

An interpretation I *satisfies* E when $I(E) = true$.

According to RDF 1.1 Semantics report:

`https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/`

RDF graphs can be viewed as *conjunctions* of simple atomic sentences in first-order logic, where *blank nodes* are *free variables* which are *understood to be existential*.

A graph G *entails* a graph E when every interpretation which satisfies G also satisfies E.

Inference

Any *process* which constructs a graph E from some other graph S is *valid* if S entails E in every case; otherwise *invalid*.

Correct and complete inference

Correct and complete *inference processes* exist for *proving* entailment of RDF graphs. This provides suitable techniques to *reason* about the semantic web.

The challenge

The semantic web as a *web of knowledge* rather than a *web of documents*

Logic has *fertilized* Computer Science from the beginning

Logic brought many *mathematicians, engineers, physicists, biologists...* to Computer Science

Logic has *inspired* computer scientists in so many different ways

Logic has *fertilized* Computer Science from the beginning

Logic brought many *mathematicians, engineers, physicists, biologists...* to Computer Science

Logic has *inspired* computer scientists in so many different ways

We can say: *Logic is (in) the soul of Computer Science*!

Encourage yourself and your students to *get deep(er)* into logic!

Encourage the Dean of your School to take logic *seriously*!

Prevent logic from *disappearing* of the academic curriculum!

Encourage yourself and your students to *get deep(er)* into logic!

Encourage the Dean of your School to take logic *seriously*!

Prevent logic from *disappearing* of the academic curriculum!

Keep Computer Science *alive* and *healthy*!

Thanks!