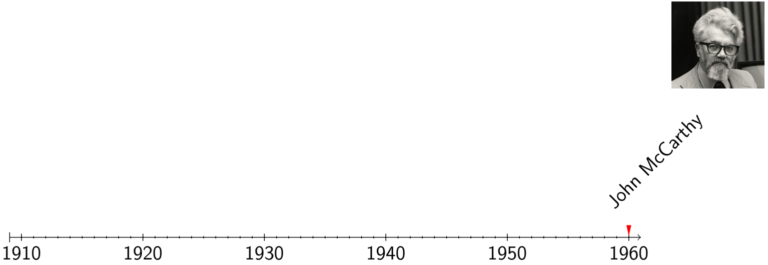
The Shoulders of Giants or Uncovering the Foundational Ideas of Lisp

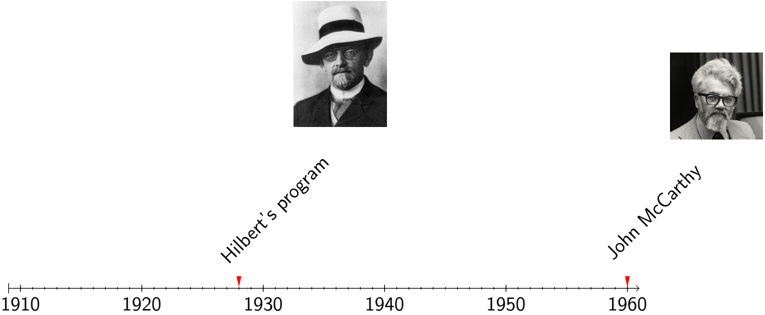
Heart of Clojure 2024

Daniel Szmulewicz

Timeline

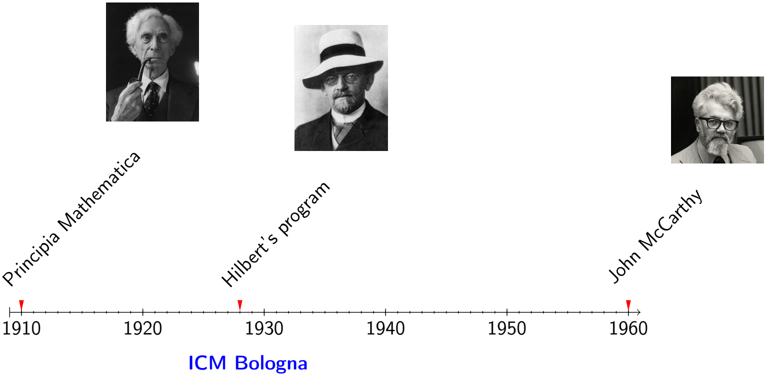


Timeline

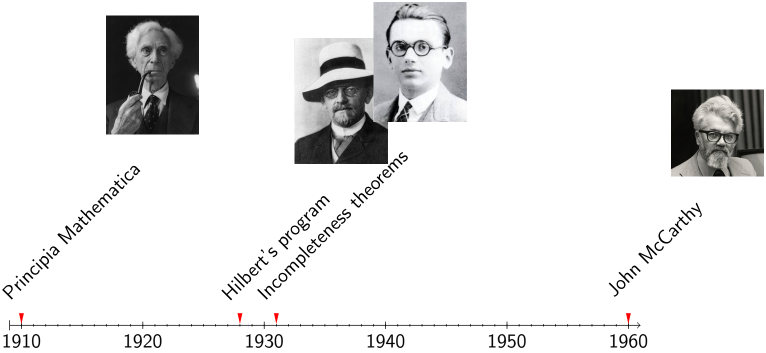


ICM Bologna

Timeline


Principia Mathematica
Hilbert's program
John McCarthy
1910
1920
1930
1940
1950
1960
ICM Bologna

Timeline



ICM Bologna

Hilbert's program

Goals

1. Show that the system is complete

Hilbert's program

Goals

1. Show that the system is complete
2. Show that the system is consistent

Hilbert's program

Goals

1. Show that the system is complete
2. Show that the system is consistent
3. Show that the system is decidable

Hilbert's program

Goals

1. ~~Show that the system is complete~~
2. Show that the system is consistent
3. Show that the system is decidable

Results

1. First incompleteness theorem

Hilbert's program

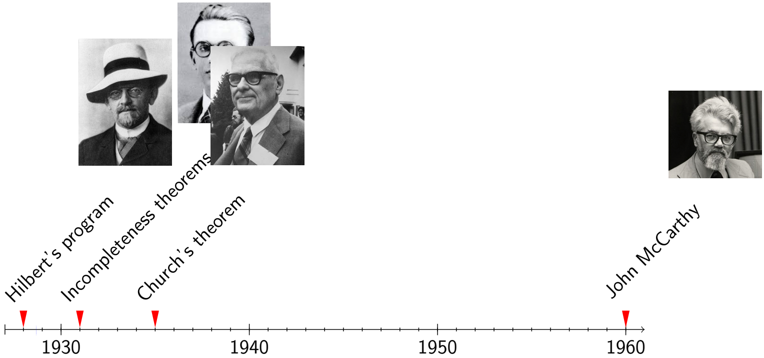
Goals

1. ~~Show that the system is complete~~
2. ~~Show that the system is consistent~~
3. Show that the system is decidable

Results

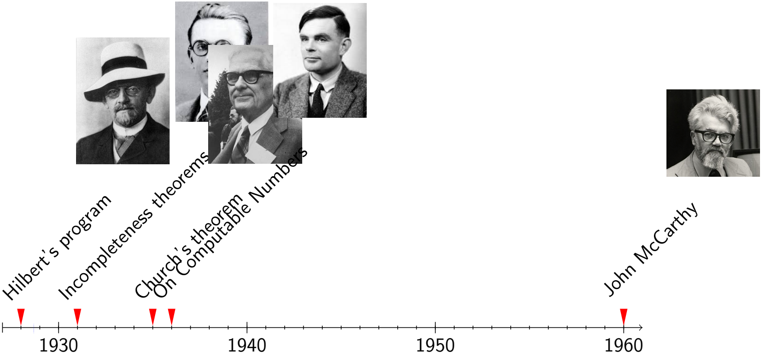
1. First incompleteness theorem
2. Second Incompleteness theorem

Timeline



ICM Bologna

Timeline



ICM Bologna

Hilbert's program

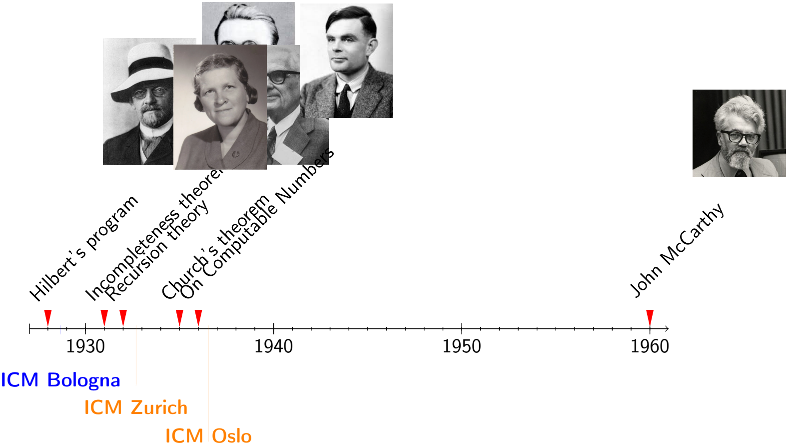
Goals

1. ~~Show that the system is complete~~
2. ~~Show that the system is consistent~~
3. ~~Show that the system is decidable~~

Results

1. First incompleteness theorem
2. Second Incompleteness theorem
3. Church's Theorem and Turing's proof

Timeline


Hilbert's program
Incompleteness theore
Recursion theory
Church's theorem
On Computable Numbers
John McCarthy
1930
1940
1950
1960
ICM Bologna
ICM Zurich
ICM Oslo

Rósza Péter

Primitive recursion: basic functions

Functions

- Zero function
- Successor function
- Projection function

Primitive recursion: basic functions

Functions

- Zero function
- Successor function
- Projection function

Clojure

```
(def Z (constantly 0))
```

Primitive recursion: basic functions

Functions

- Zero function
- Successor function
- Projection function

Clojure

```
(def Z (constantly 0))
(def S inc)
```

Primitive recursion: basic functions

Functions

- Zero function
- Successor function
- Projection function

Clojure

```
(def Z (constantly 0))
(def S inc)
(defn P [i]
  (fn [& args]
    (nth args (dec i))))
```

Primitive recursion: operations

Operations

- Composition
- Recursion

Clojure

```
(def C comp) ; approximation
```

Primitive recursion: operations

Operations

- Composition
- Recursion

Clojure

```
(def C comp) ; approximation
(def R recur) ; approximation
```

Class of primitive recursive functions

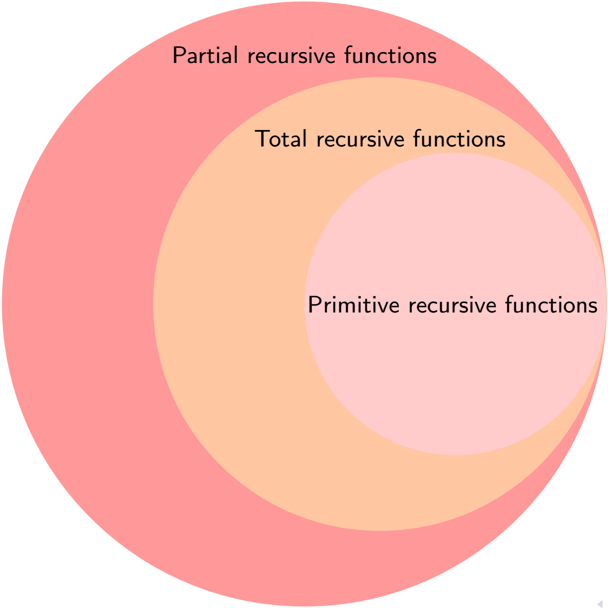
Numeric operations

- addition
- subtraction
- multiplication
- division
- modulo
- return the nth prime
- exponentiation
- factorial
- distance
- maximum
- minimum

Propositional calculus

- negation
- boolean
- conjunction
- disjunction
- conditional

Visualization



Total recursive functions

Ackermann function

```
(defn ackermann [m n]
  (cond (zero? m) (inc n)
        (zero? n) (ackermann (dec m) 1)
        :else (ackermann (dec m) (ackermann m (dec n)))))
```

Partial recursive functions

Functions

- Zero function
- Successor function
- Projection function

Operations

- Composition
- Recursion
- Minimisation

McCarthy formalism

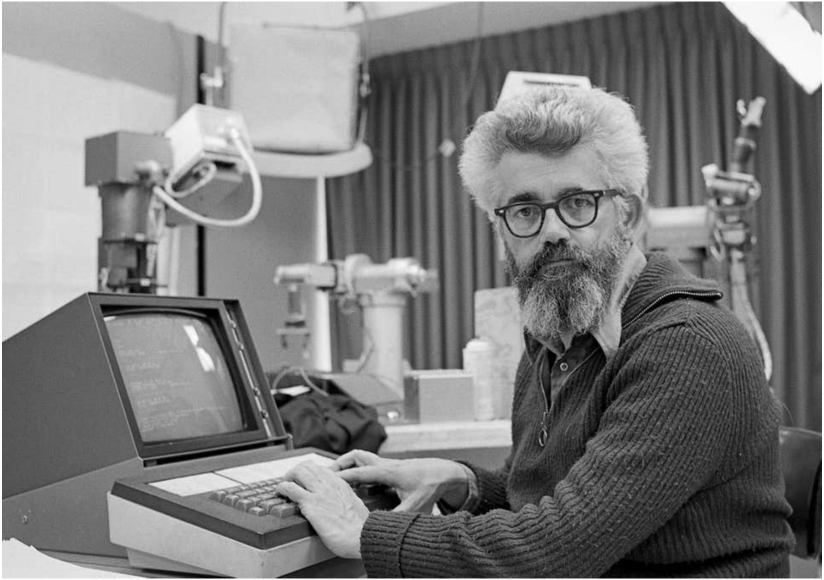
Functions

- Zero function
- Successor function
- Projection function

Operations

- Composition
- Recursive function definitions
- IF-THEN-ELSE

John McCarthy



Lambda calculus I

To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church (1941). I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions. (McCarthy, John, 1978)

Now, having borrowed this notation, one of the myths concerning LISP that people think up or invent for themselves becomes apparent, and that is that LISP is somehow a realization of the lambda calculus, or that was the intention. The truth is that I didn't understand the lambda calculus, really. (McCarthy, John, 1978)

Turing machines I

In the early days of computing, some people developed programming languages based on Turing machines; perhaps it seemed more scientific. Anyway, I decided to write a paper describing LISP both as a programming language and as a formalism for doing recursive function theory. (McCarthy, John, 1978)

Turing machines II

Another way to show that Lisp was neater than Turing machines was to write a universal Lisp function and show that it is briefer and more comprehensible than the description of a universal Turing machine. This was the Lisp function EVAL. (McCarthy, John, 1978a)

Eval

S.R. Russell noticed that eval could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter. (McCarthy, John, 1978)

Mathematical notation

Math notation

$$C_n^k(x_1, \ldots, x_k) \stackrel{\text{def}}{=} 0 \tag{1}$$

$$S(x) \stackrel{\text{def}}{=} x + 1 \tag{2}$$

$$P_i^k(x_1, \ldots, x_k) \stackrel{\text{def}}{=} x_i \tag{3}$$

Clojure

```
(def Z #(fn [& _] 0))
(def S inc)
(defn P [i]
  (fn [& args] (nth args (dec i))))
```

Teaser

```
(defn foo [n]
  (fn [f h r]
    (add (mul (f n) (bool (r n))) (mul (h n) (not (r n))))))
```

Hint: a tribute to McCarthy

In Clojure parlance

Addition

```
(defn add [x y]
  (let [f (P 1)
           g (C S (P 2))]
    ((R x y) f g)))
```

Usage

```
(add 3 4)
```

7

In Clojure parlance

Multiplication

```
(defn mul [x y]
  (let [f (Z)
        g (C add (P 2) (P 3))]
    ((R x y) f g)))
```

Usage

```
(mul 3 4)
```

12

Logical NOT

Negation

```
(defn not [x]
  (let [f (constantly 1)
        g (Z)]
    ((R x) f g)))
```

Usage

```
[(not 0) (not 1) (not 2)]
```

| 1 | 0 | 0 |

Boolean

Boolean

```
(defn bool [x]
  (let [f (Z)
        g (k 1)]
    ((R x) f g)))
```

Usage

```
[(bool 0) (bool 1) (bool 2)]
```

| 0 | 1 | 1 |

Recursion

```
(defn R [n & xs]
  (fn [f g]
    (loop [i 1
           j 0
           acc (apply f xs)]
      (if (<= i n)
        (recur (inc i) (inc j) (apply
                                 (partial g j acc)
                                xs))
        acc))))
```