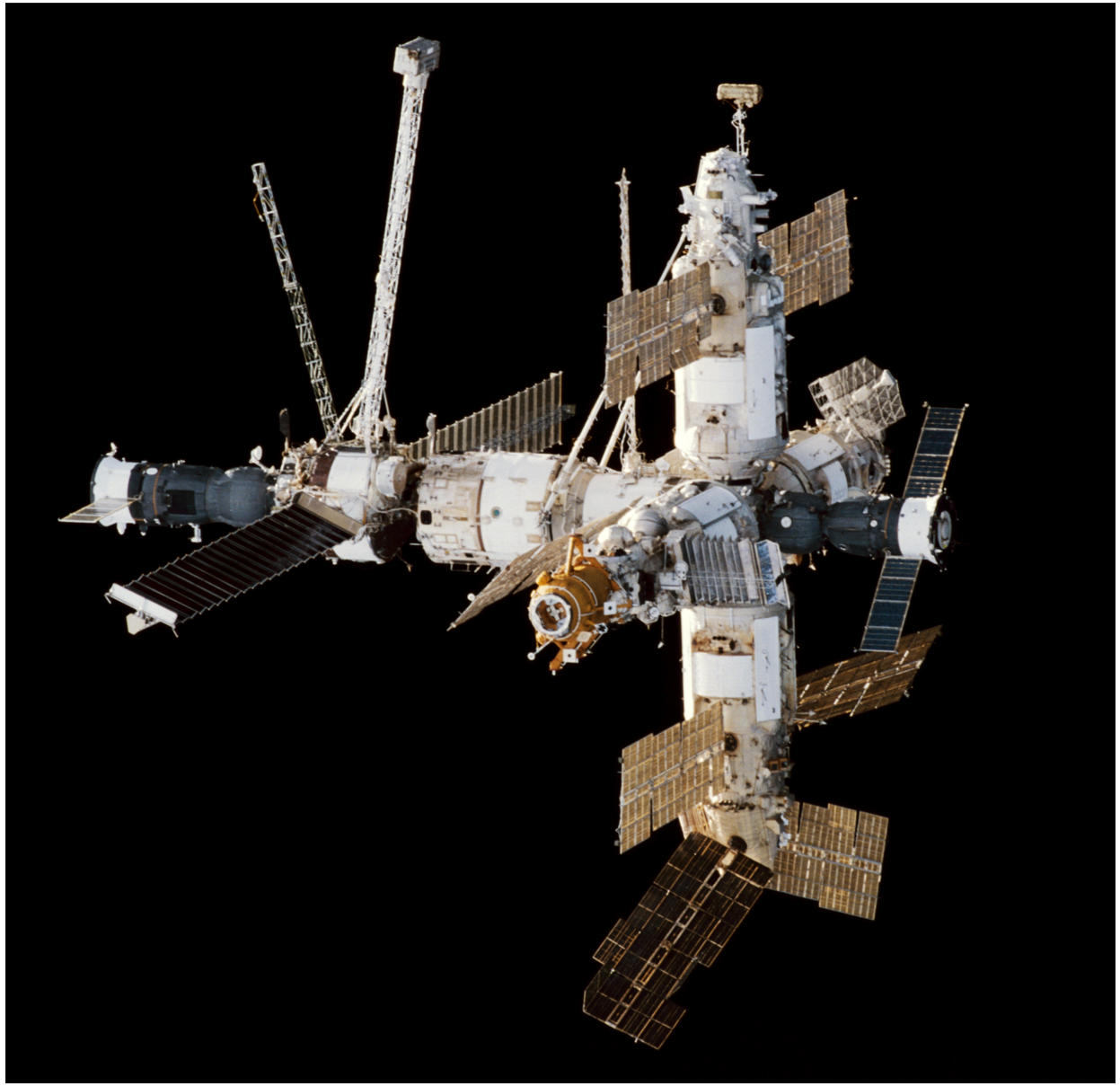
1986


C/2012 S1 (ISON)
2013-11-15 12:08 UTC
D. Peach

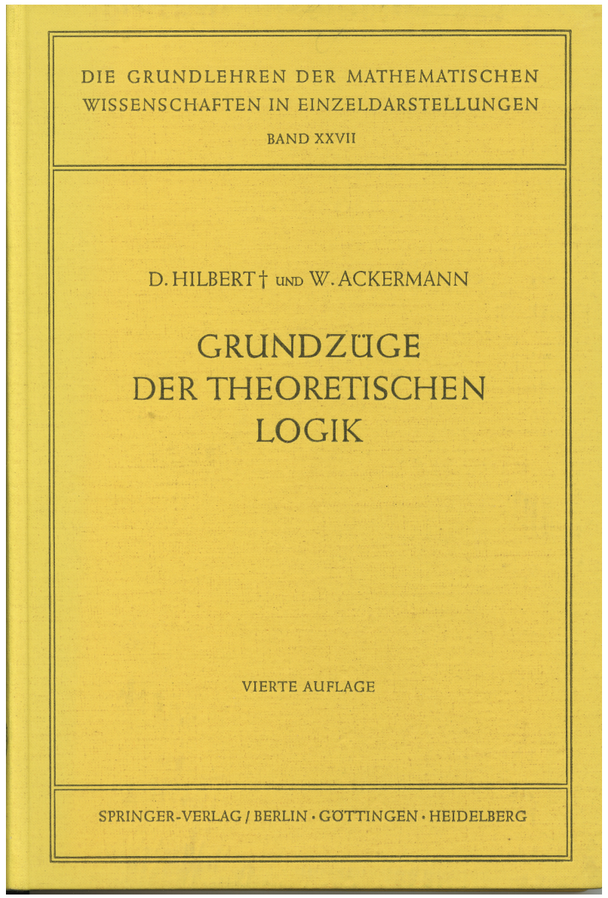



"MAVERICK"

Programming Language	1986	1991	1996	2001	2006	2011	2016
Java	–	–	15	2	1	1	1
C	1	1	1	1	2	2	2
C++	5	2	2	3	3	3	3
C#	–	–	–	10	6	5	4
Python	–	–	24	23	7	6	5
PHP	–	–	–	8	4	4	6
JavaScript	–	–	19	7	8	9	7
Visual Basic .NET	–	–	–	–	–	28	8
Perl	–	–	3	4	5	8	9
Ruby	–	–	–	31	17	10	10
Lisp	3	4	7	15	12	12	27
Ada	2	5	6	16	15	16	28
Pascal	6	3	4	13	16	14	74

David Hilbert

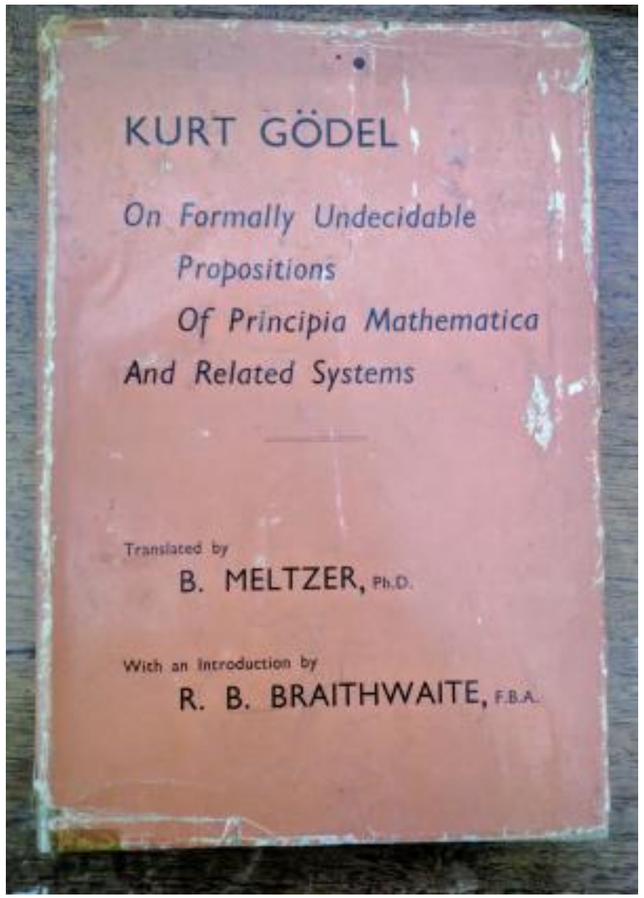
David Hilbert



David Hilbert

Kurt Gödel

Kurt Gödel


KURT GÖDEL
On Formally Undecidable
Propositions
Of Principia Mathematica
And Related Systems
Translated by
B. MELTZER, Ph.D.
With an Introduction by
R. B. BRAITHWAITE, F.B.A.

"This statement is not provable"

Is the Entscheidungsproblem decidable?

What is decidable?

Alonzo Church

Alonzo Church

We shall use heavy type letters to stand for variable or undetermined formulas. And we adopt the convention that, unless otherwise stated, each heavy type letter shall represent a well-formed formula and each set of symbols standing apart which contains a heavy type letter shall represent a well-formed formula.

When writing particular well-formed formulas, we adopt the following abbreviations. A formula $\{\boldsymbol{F}\}(\boldsymbol{X})$ may be abbreviated as $\boldsymbol{F}(\boldsymbol{X})$ in any case where $\boldsymbol{F}$ is or is represented by a single symbol. A formula $\{\{\boldsymbol{F}\}(\boldsymbol{X})\}(\boldsymbol{Y})$ may be abbreviated as $\{\boldsymbol{F}\}(\boldsymbol{X}, \boldsymbol{Y})$, or, if $\boldsymbol{F}$ is or is represented by a single symbol, as $\boldsymbol{F}(\boldsymbol{X}, \boldsymbol{Y})$. And $\{\{\{\boldsymbol{F}\}(\boldsymbol{X})\}(\boldsymbol{Y})\}(\boldsymbol{Z})$ may be abbreviated as $\{\boldsymbol{F}\}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z})$, or as $\boldsymbol{F}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z})$, and so on. A formula $\lambda \boldsymbol{x}_1[\lambda \boldsymbol{x}_2[\cdots \lambda \boldsymbol{x}_n[\boldsymbol{M}]\cdots]]$ may be abbreviated as $\lambda \boldsymbol{x}_1 \boldsymbol{x}_2 \cdots \boldsymbol{x}_n \cdot \boldsymbol{M}$ or as $\lambda \boldsymbol{x}_1 \boldsymbol{x}_2 \cdots \boldsymbol{x}_n \boldsymbol{M}$.

We also allow ourselves at any time to introduce abbreviations of the form that a particular symbol α shall stand for a particular sequence of symbols $\boldsymbol{A}$, and indicate the introduction of such an abbreviation by the notation $\alpha \rightarrow \boldsymbol{A}$, to be read, "$\alpha$ stands for $\boldsymbol{A}$."

We introduce at once the following infinite list of abbreviations,

$$1 \rightarrow \lambda ab \cdot a(b),$$
$$2 \rightarrow \lambda ab \cdot a(a(b)),$$
$$3 \rightarrow \lambda ab \cdot a(a(a(b))),$$

and so on, each positive integer in Arabic notation standing for a formula of the form $\lambda ab \cdot a(a(\cdots a(b)\cdots))$.

The expression $\mathrm{S}^{\boldsymbol{x}}_{\boldsymbol{N}}\boldsymbol{M}$ | is used to stand for the result of substituting $\boldsymbol{N}$ for $\boldsymbol{x}$ throughout $\boldsymbol{M}$.

We consider the three following operations on well-formed formulas:

Kurt Gödel

Kurt Gödel

General recursive functions of natural numbers[1]).

Von

S. C. Kleene in Madison (Wis., U.S.A.).

The substitution

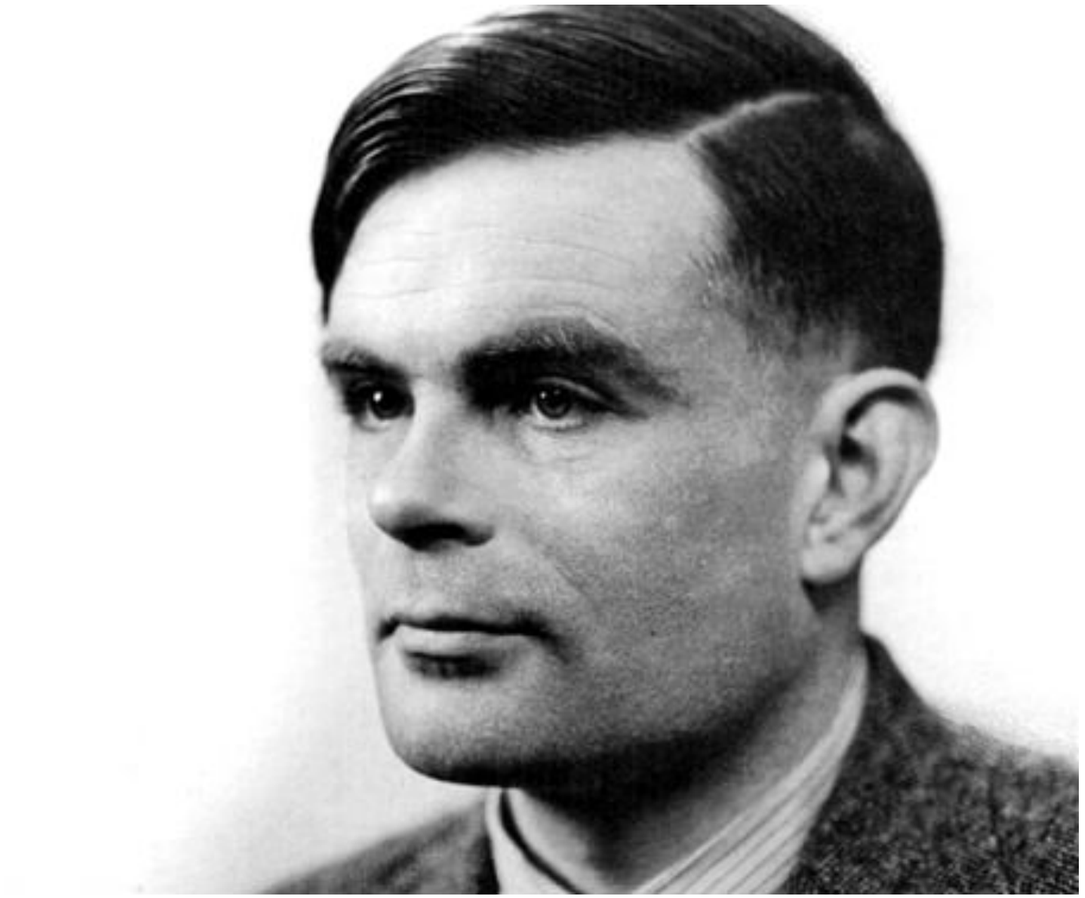
1) $$\varphi(x_1, \ldots, x_n) = \theta(\chi_1(x_1, \ldots, x_n), \ldots, \chi_m(x_1, \ldots, x_n)),$$

and the ordinary recursion with respect to one variable

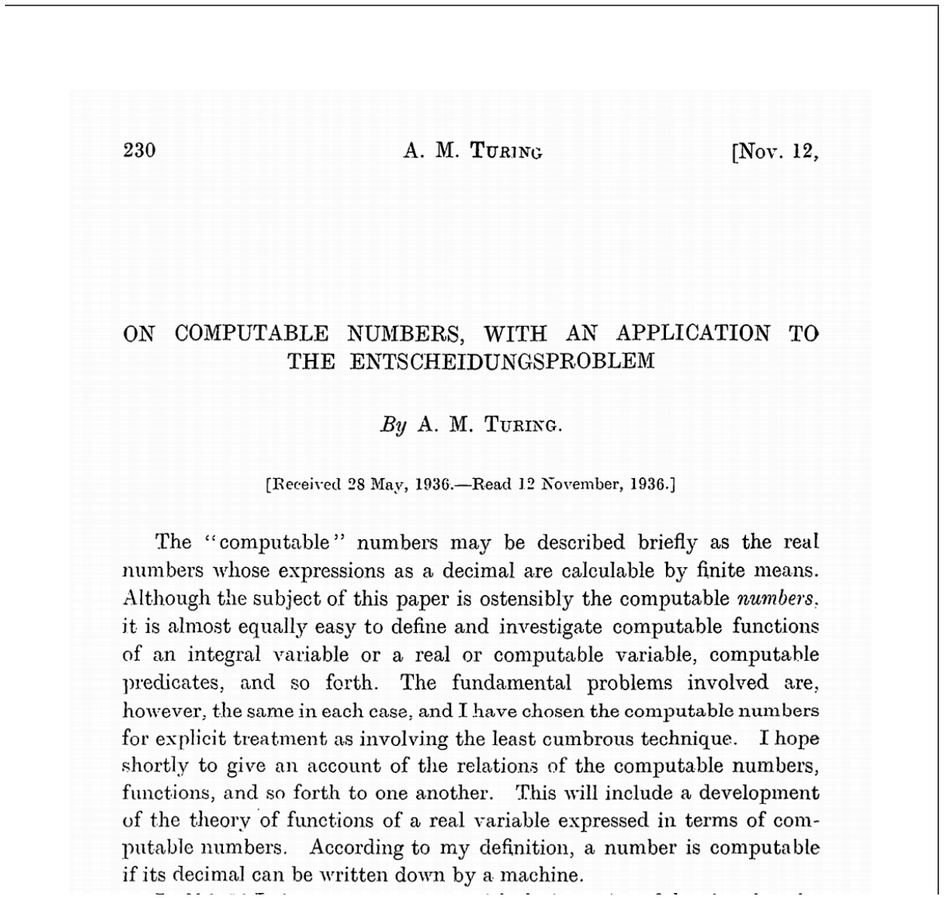
(2) $$\varphi(0, x_2, \ldots, x_n) = \psi(x_2, \ldots, x_n)$$
$$\varphi(y+1, x_2, \ldots, x_n) = \chi(y, \varphi(y, x_2, \ldots, x_n), x_2, \ldots, x_n),$$

where $\theta, \chi_1, \ldots, \chi_m, \psi, \chi$ are given functions of natural numbers, are examples of the definition of a function φ by equations which provide a step by step process for computing the value $\varphi(k_1, \ldots, k_n)$ for any given set $k_1, \ldots, k_n$ of natural numbers. It is known that there are other definitions of this sort, e. g. certain recursions with respect to two or more variables simultaneously, which cannot be reduced to a succession of substitutions and ordinary recursions[2]). Hence, a characterization of the notion of recursive definition in general, which would include all these cases, is desirable. A definition of general recursive function of natural numbers was suggested by Herbrand to Gödel, and was used by Gödel with an important modification in a series of lectures at Princeton in 1934. In this paper we offer several observations on general recursive functions, using essentially Gödel's form of the definition.

Alan Turing



Alan Turing



230 A. M. TURING [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

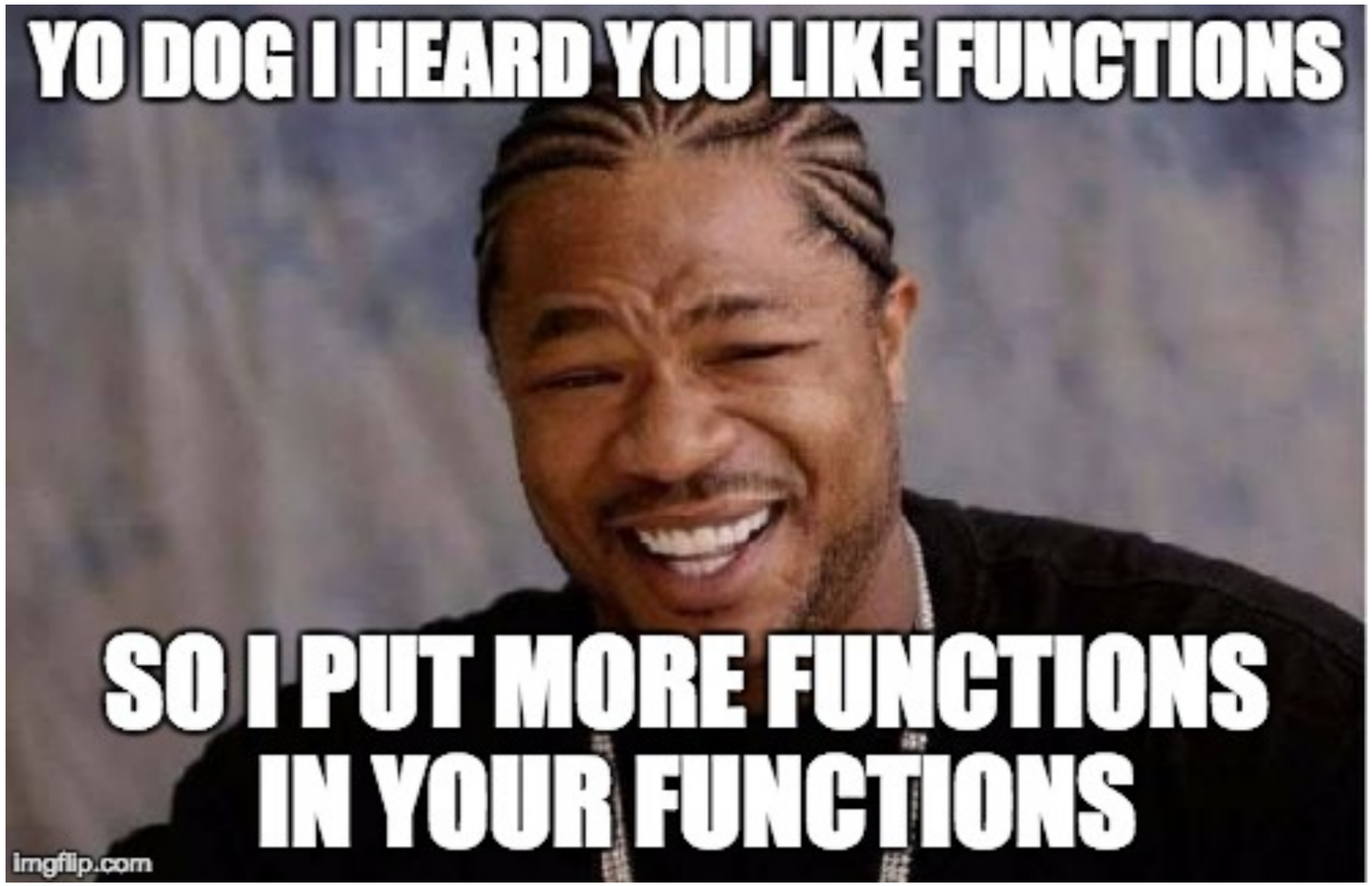
Church-Turing equivalence

One problem, three equivalent solutions

Lambda Calculus

```
L, M, N :=    x
          |   (lambda x. N)
          |   (L M)
```

Booleans? If? Else? NUMBERS??



Booleans

```
true x y =
      x

false x y =
      y
```

If then else

```
ifte bool t e =
     bool t e
```

And

```
and p q =
    p q p

and true false =
    true false true

and false true =
    false true false

…
```

Or

```
or p q =
   p p q

or true false =
   true true false

or false false =
   false false false
```

We can build any construct with lambda calculus

But how do we encode numbers?

```
zero  f x = x
one   f x = f x
two   f x = f ( f x )
three f x = f ( f ( f x ) )
...


> two (+1) 0
2
```

Addition

```
add m n f x = m f (n f x)
```

```
> add one two (+1) 0
3
```

Multiplication

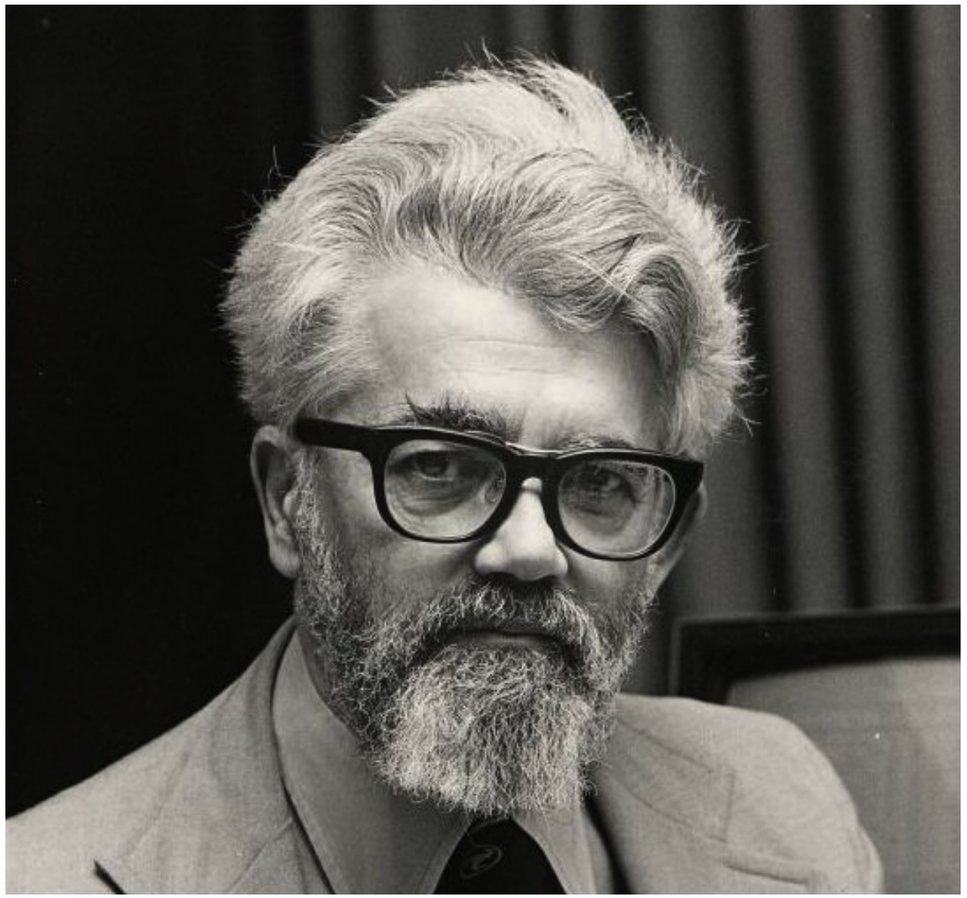
```
mul m n f x = m (n f) x
```

```
> add one (mul two two) (+1) 0
5
```

Remember, this is the 1940's

- Computers were actual people
- No compilers
- No programming language

John McCarthy (1960's)



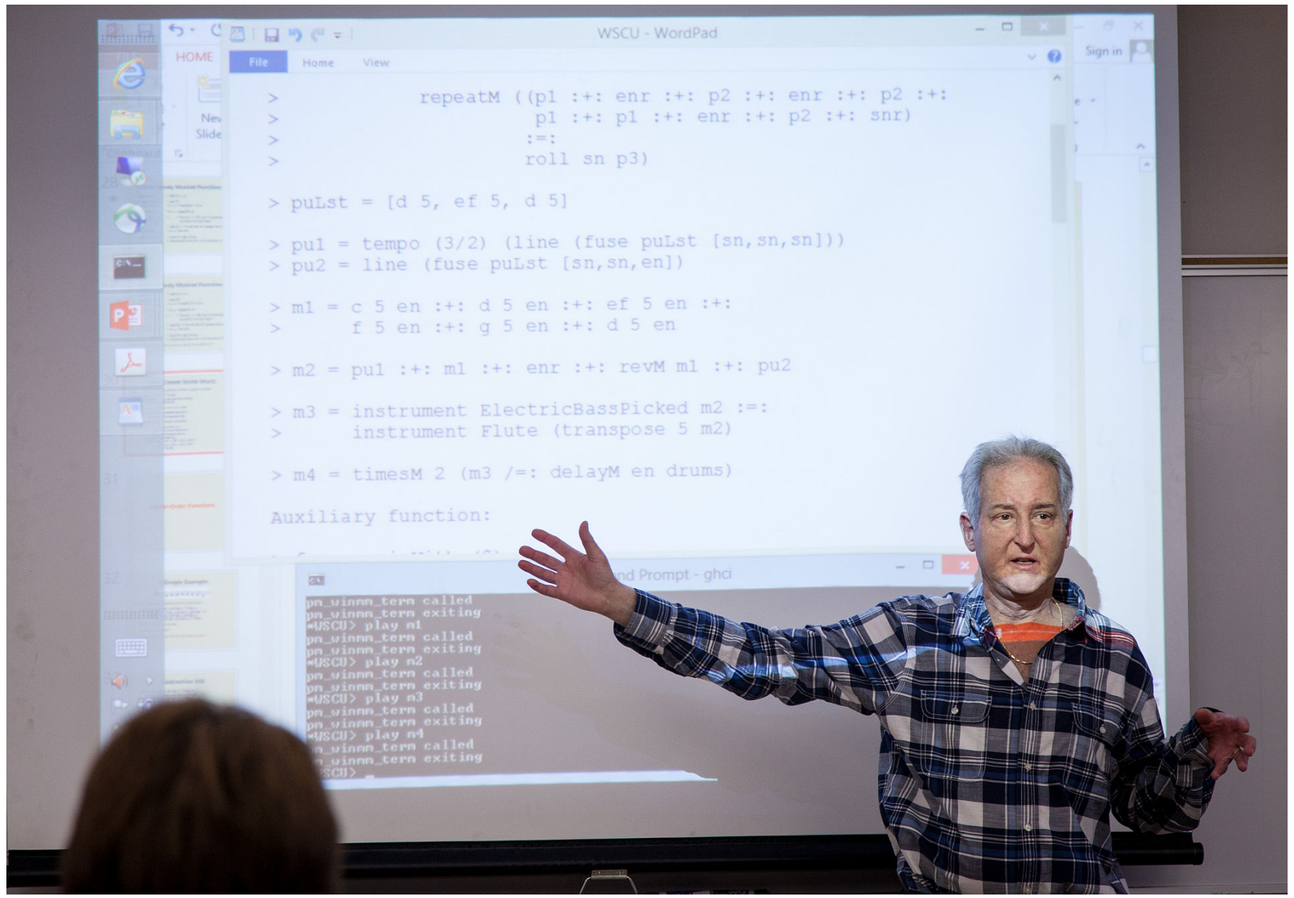
LISP

```
(LABEL FACT (LAMBDA (N)
    (COND ((ZEROP N) 1)
          (T (TIMES N (FACT SUB1 N))))))
```

1960s - 1990s

- Lots of programming languages
- Many them were functional
- Little collaboration

1990s - Paul Hudak



Haskell Curry

Haskell (1990)

Lazy evaluation

The value is infinite

```
> take 20 [1..]

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

Mapping on inifite list

```
> take 20 $ map (+1) [1..]

[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]
```

All the iterations of a function

```
iterate f x = [x, f x, f (f x), ...]

take 5 $ iterate (*3) 1

[1,3,9,27,81]
```

Expressiveness

How to model UI like this?

Expressiveness

- Inifinite list of events
- Define functions to be applied to those events
- Result is an infinite list of side effects

Advantages

- No callbacks
- Pure functions
- Handling time variant instead of event based

Elm

```
lift : (a -> b) -> Signal a -> Signal b
```

```
lift isConsonant Keyboard.lastPressed
```

Elm

```
foldp : (a -> b -> b) -> b -> Signal a -> Signal b
```

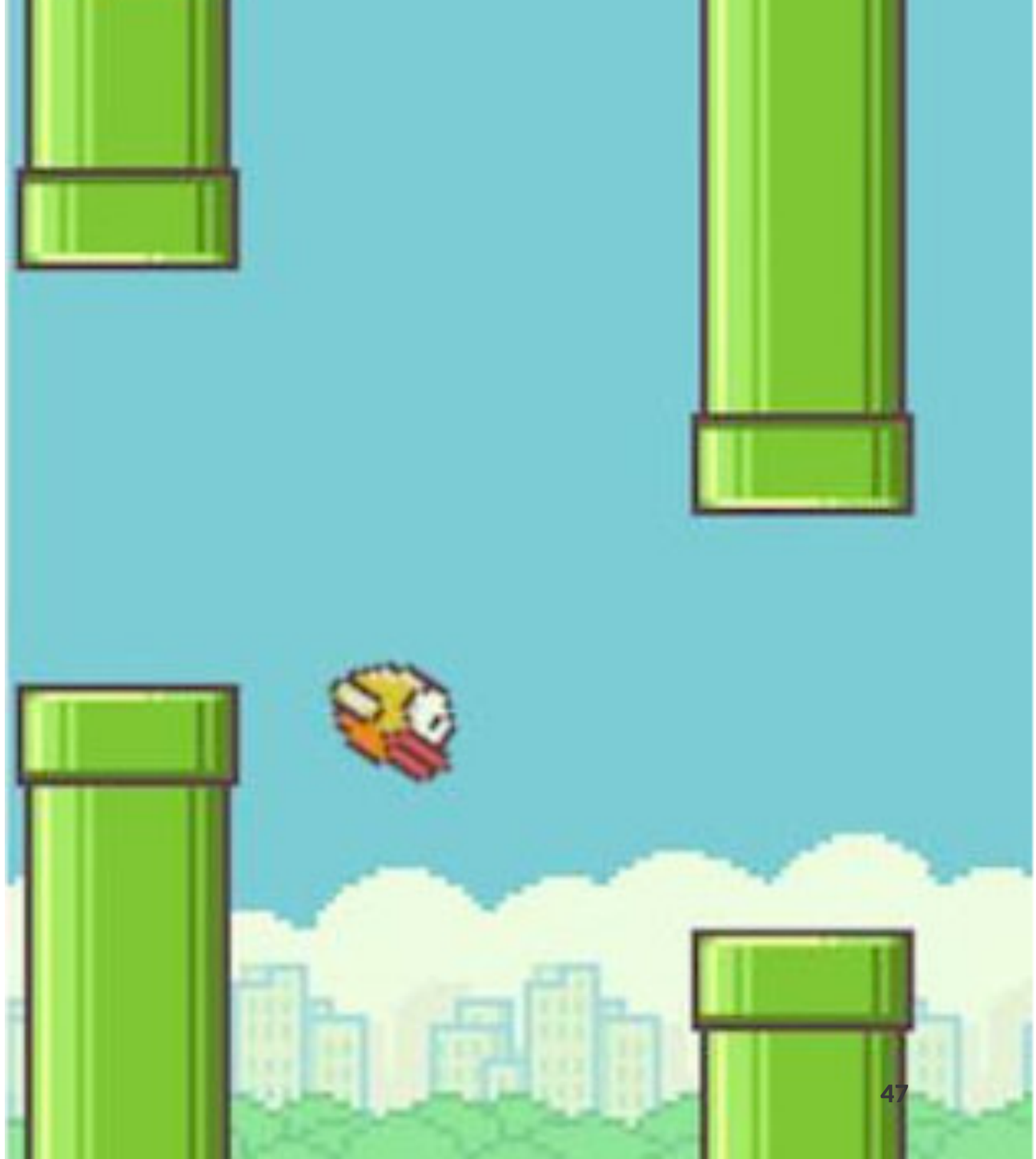
```
foldp (\key count -> count + 1) 0 Keyboard.lastPressed
```

Since Elm 0.17, there are no more signals

Farewell Functional Reactive Programming: http://elm-lang.org/blog/farewell-to-frp

Let's build a Flappy Bird game

- Generate pipes every X seconds
- Handle the key pressed event
- Handle colision
- Have a live scoreboard



Elm Subscriptions

- Replacement of Signals
- Applying functions on infinite streams
- Generate messages

Elm Subscriptions

```
subscriptions : Game -> Sub Msg
subscriptions model =
    Sub.batch
        [ AnimationFrame.diffs TimeUpdate
        , Keyboard.downs KeyDown
        , Time.every Time.second AskForTopPlayers
        , Time.every (Time.second * 2) GeneratePipe
        , Phoenix.Socket.listen model.phxSocket PhoenixMsg
        ]
```

Elm Subscriptions

```
subscriptions : Game -> Sub Msg
subscriptions model =
    Sub.batch
        [ AnimationFrame.diffs TimeUpdate
        , Keyboard.downs KeyDown
        , Time.every Time.second AskForTopPlayers
        , Time.every (Time.second * 2) GeneratePipe
        , Phoenix.Socket.listen model.phxSocket PhoenixMsg
        ]
```

Elm Subscriptions

```
subscriptions : Game -> Sub Msg
subscriptions model =
    Sub.batch
        [ AnimationFrame.diffs TimeUpdate
        , Keyboard.downs KeyDown
        , Time.every Time.second AskForTopPlayers
        , Time.every (Time.second * 2) GeneratePipe
        , Phoenix.Socket.listen model.phxSocket PhoenixMsg
        ]
```

Elm

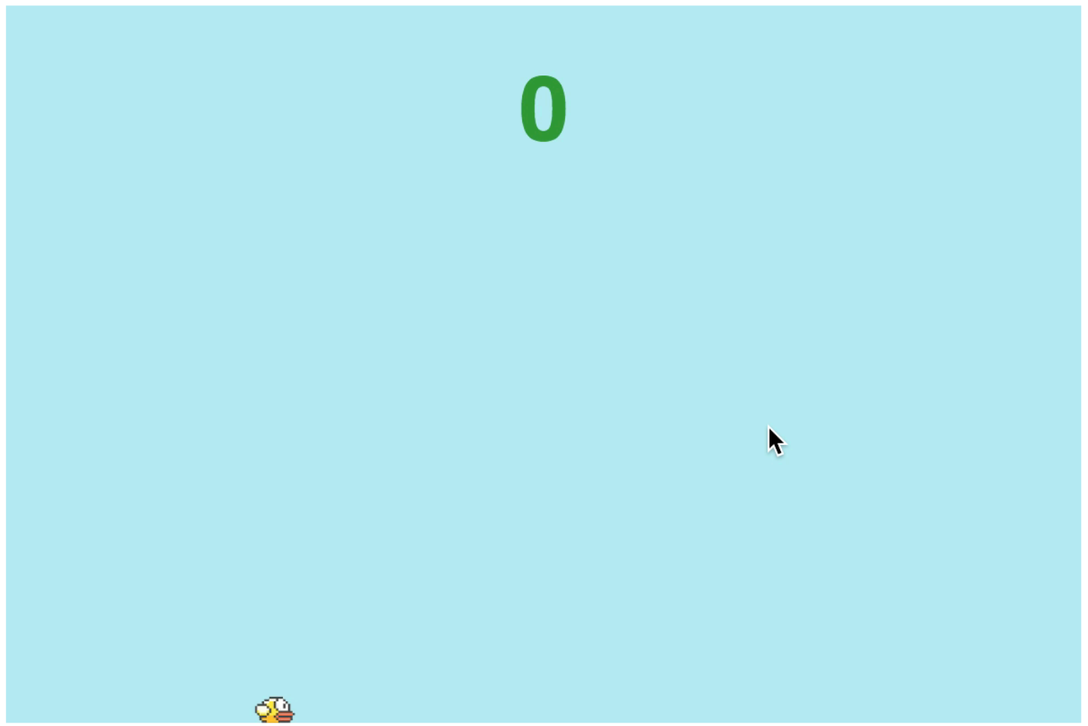
- Subscriptions generate messages
- User can generate messages (click button)
- State transformation through messages

Elm

```
update : Msg -> Game -> ( Game, Cmd Msg )
update msg game =
    case game.state of
        Play ->
            case msg of
                AskForTopPlayers _ ->
                  ...
                SendScore ->
                  ...
                KeyDown keyCode ->
                  ...
                GeneratePipe _ ->
                  ...
```

Elm

```
view : Game -> Html Msg
view model =
    div []
        [ text (toString game) ]
```


0

Summary

- Functional programming is invented, not discovered
- Are you using a programming language that is invented?
- Expressiveness by design

Thank you

Twitter: @paulodiniz

Github: paulodiniz

I like beers, let's have one after! 🍺