A Seminar Presentation on

Recursiveness, Computability and The Halting Problem

Presented By :

Abhinav Aggarwal

CSI-IDD, V^{th} yr

Indian Institute of Technology Roorkee

Joint work with:

Prof. Padam Kumar

A quick glance…

- ✓ Nature of computation
- ✓ Classical Recursion Theory – **Theory of functions and sets of natural numbers**
- ✓ Goes back to Dedekind – defining functions using recurrence
- ✓ Recursiveness – Church, Gödel, Kleene, Turing, Post
- ✓ Effectively computable functions = Recursive functions
- ✓ Development of programming languages
- ✓ Termination of Programs

Recursiveness

Primary references :

[1] Odifreddi, P., "*Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*", Elsevier Science, 1980

[2] Post, "*Absolutely unsolvable problems and relatively undecidable propositions*", The Undecidable, Raven Press, 1965, pp. 265 – 283

[3] Gödel, "*On undecidable propositions of formal mathematical systems*", The Undecidable, Raven Press, 1965, pp. 41 – 81

[4] Kleene, "*General recursive functions of natural numbers*", The Undecidable, Raven Press, 1965, pp. 237 – 252

[5] Turing, A. M., "*On computable numbers with an application to the Entscheidungsproblem*", Proc. London Math. Soc. 42, 1936, pp. 230 – 265

A new word …

- ✓ *Recursion, recurrence* – perhaps they were already taken
- ✓ Closer look at functions from ω to ω
  $\omega = \{0,1,2,3,...\}$
- ✓ Characterization of natural numbers – **Dedekind's formalization**

0, S(0), S(S(0)), S(S(S(0))), …

- ✓ Formal arithmetic models – loads of axioms

Axioms I.1.1 (Dedekind [1888])

A1 $S(x) = S(y) \rightarrow x = y$

A2 $0 \neq S(y)$

A3 $x \neq 0 \rightarrow (\exists y)(x = S(y))$.

$$\begin{aligned} \mathcal{O}(x) &= 0 \\ S(x) &= x + 1 \\ \mathcal{I}_i^n(x_1, \ldots, x_n) &= x_i \qquad (1 \leq i \leq n) \end{aligned}$$

Initial Functions

Where does it take us?

- ✓ Dedekind's Induction Principle

$$\varphi(0) \wedge (\forall x)[\varphi(x) \rightarrow \varphi(S(x))] \rightarrow (\forall y)\varphi(y).$$

- ✓ Least Number Principle

$$(\exists y)\psi(y) \rightarrow (\exists z)[\psi(z) \wedge (\forall x < z)\neg\psi(x)].$$

- ✓ Primitive Recursive functions

$$\begin{aligned} f(\vec{x}, 0) &= g(\vec{x}) \\ f(\vec{x}, y+1) &= h(\vec{x}, y, f(\vec{x}, y)). \end{aligned}$$

- ✓ μ-recursive functions

$$(\forall \vec{x})(\exists y)(g(\vec{x}, y) = 0)$$

$$f(\vec{x}) = \mu y(g(\vec{x}, y) = 0).$$

- ✓ Class of recursive functions

Examples …

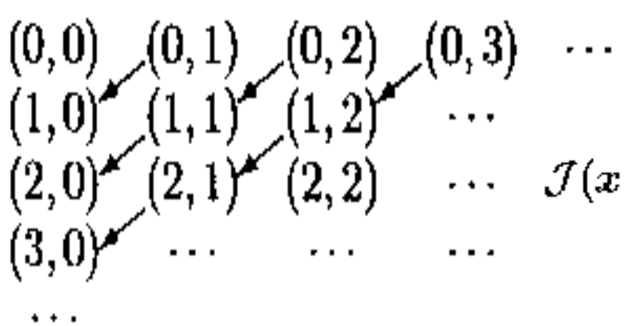
- ✓ Commonly encountered functions are recursive – *bounded sum, bounded product, factorial, prime number generation*

f(x,y) = x + y

$$\begin{aligned} f(x,0) &= x \\ f(x,y+1) &= S(f(x,y)). \end{aligned}$$

$$\begin{aligned} h(x,y,z) &= S(\mathcal{I}_3^3(x,y,z)) \\ f(x,0) &= \mathcal{I}_1^1(x) \\ f(x,y+1) &= h(x,y,f(x,y)). \end{aligned}$$

- ✓ Coding (numbering) of the entities of a set – *Countability is recursive* – **Cantor's coding of the plane of ordered pairs**



$$\mathcal{J}(x,y) = \Big(\sum_{i<x+y}(i+1)\Big) + x = \frac{(x+y)^2+3x+y}{2}$$

Curious Bob ...

- ✓ Is every function recursive? – **No**
- ✓ What about mathematical functions? - **Yes**
- ✓ But where is the "re-occurrence"? - **In the initial functions**
- ✓ OK. But why can't I compute them directly, instead of using recursion? – **Formal arithmetic**
- ✓ How is it useful at all? Doesn't it complicate things? – **It gives them a structure – Facilitates generalization**
- ✓ Fine. But how do I know if this will work? Do you have a magic wand? - **Computability**

Computability

Primary references :

[1] Odifreddi, P., "*Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*", Elsevier Science, 1980

[2] Soare, R. I., "*Computability and Recursion*", 10th Int. Cong. for Logic, Methodology and Philosophy of Science, Sec. 3 : Recursion Theory and Constructivism, 1995

A **function** is "*computable*" (also called "*effectively calculable*" or simply "*calculable*") if it can be calculated by a **finite mechanical procedure**.

$$f(x)=\begin{cases} 1 & x=0 \\ f(\lfloor x-1 \rfloor)+1 & x>0 \end{cases} \qquad g(x)=\begin{cases} 1 & x=0 \\ g(\lfloor x+1 \rfloor)+1 & x>0 \end{cases}$$

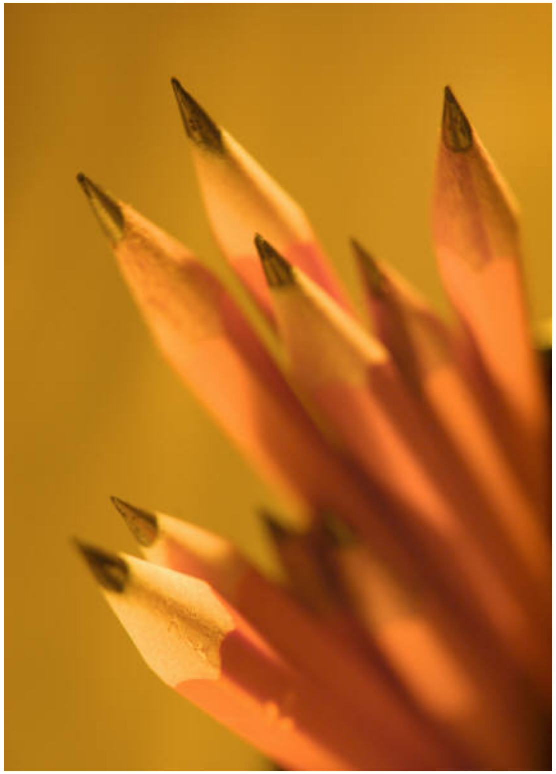
Can we **devise** an **algorithm** for the given function (task) and **check** if it can be implemented using the **resources** (theoretical or practical) **available at that time**?

Code – C/C++, Java, PASCAL, Haskell, Scheme, LISP

Algorithm – Flowcharts, Pseudocode, Textual Description

Mathematical Models – Turing Machine, Lambda expression

The first thought …



- ✓ Written Calculations on a sheet of paper
- ✓ Perceive what is written by reading it
- ✓ Make a decision for what to write next
- ✓ Read, Think, Write – **Turing Machine**

*A function is **Turing Machine Computable** if there is a Turing Machine which reaches a final configuration with f(**x**) represented in unary notation on its tape, when started with only **x** in unary notation on the tape.*

- ✓ Machine-dependent languages

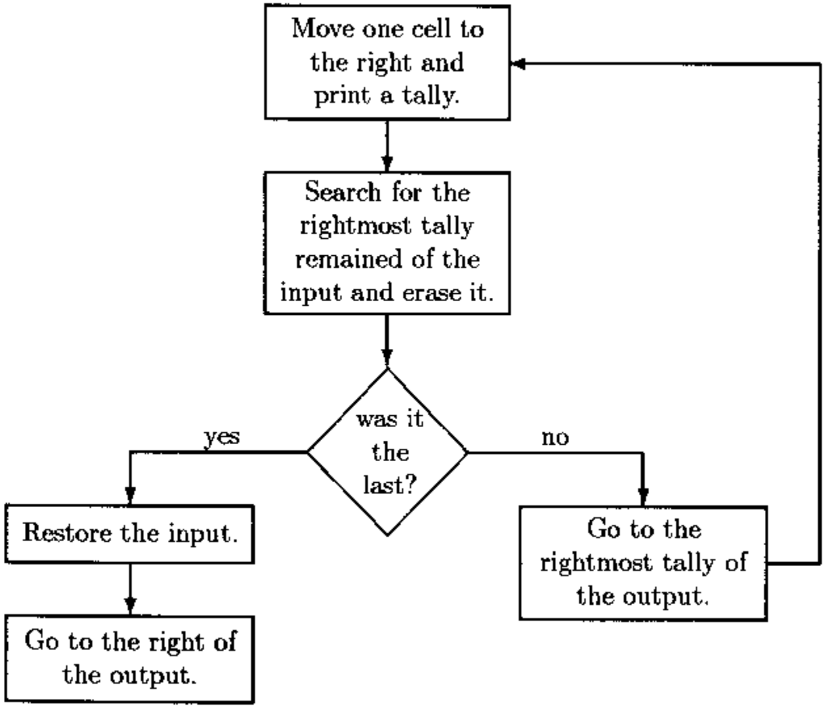


Figure I.1: Flowchart for $\mathcal{I}_1^1$

$$I_1^1(x) = x$$

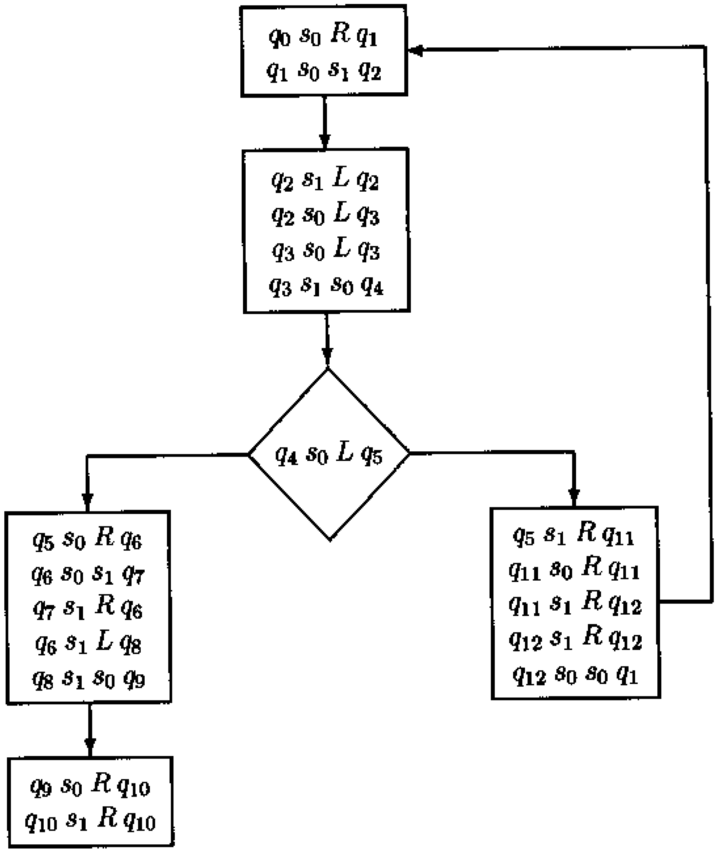


Figure I.2: Program for $\mathcal{I}_1^1$

The next step …

- ✓ Machine independent abstract core
- ✓ Emphasis on sequence of steps for a general input
- ✓ Making a flowchart for the process involved – **Flowchart Program**

*A function is **Flowchart Computable** if there exists a flowchart which modifies the input variables **x** and exits (halts) when the output variable takes the value **f(x)**.*

- ✓ General Purpose Simulation Systems
- ✓ **For-programs** and **While-programs**
- ✓ Algol, Pascal, Basic, Fortran

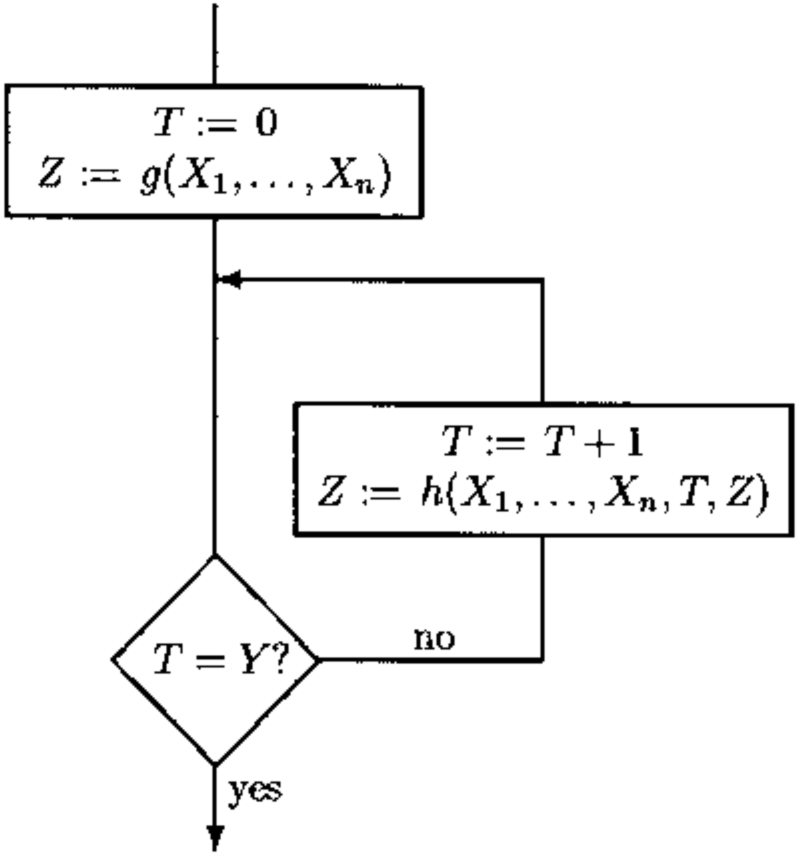
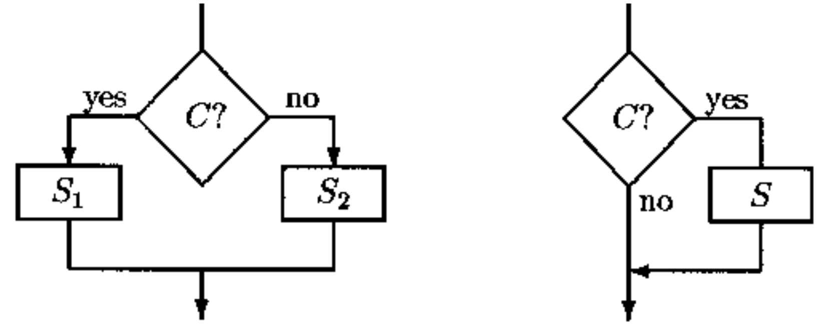
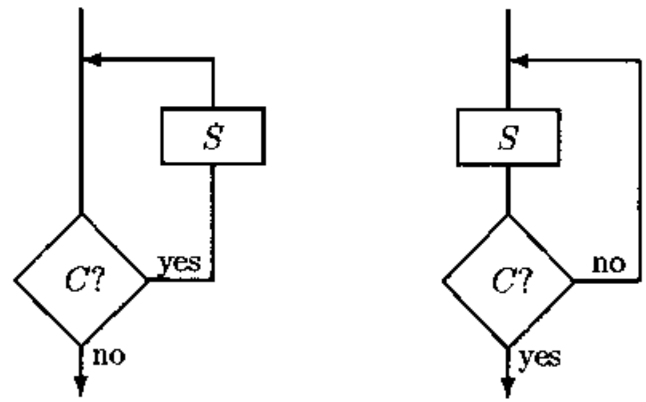


Figure I.8: Flowchart for primitive recursion



Forward conditionals (**if** C **then** S_1 **else** S_2, **if** C **then** S)



Backward conditionals (**while** C **do** S, **repeat** S **until** C)

Figure I.10: Structured flowcharts

A higher vision …

- ✓ Can functions act as arguments? – **First class citizens**
- ✓ No distinction between functions and data – **λ - abstractions**
- ✓ Any other evidence? – **Gödel Numbering**

Enter Church …

$$\mathbf{n} \equiv \lambda f.\lambda x.f^{(n)}(x)$$

Church Numerals

*A function **f** is **λ-definable** if there is a λ-term F such that f(a) = b holds if and only if F(**a**) = **b** is true, upto β-reductions.*

Functional Programming Languages
LISP, Scheme, Haskell, SML, Scala

Putting it all together …

- ✓ Church's Thesis – ***Every computable function is recursive***
- ✓ Post - General recusiveness = Effective computability
- ✓ Absolute unsolvability of functions
- ✓ Restricted versions of computability – Polynomial Time Computability
- ✓ Complexity Classes
- ✓ Equivalence and Inclusion relations between complexity classes
- ✓ Effective ***optimized*** computability

Extending a little …

- ✓ Defining functions for only some inputs – **Partial Recursive Functions**
- ✓ Kleene – *Recursive functions are exactly the partial recursive functions which happen to be total*
- ✓ Index **e** to every p.r.f – **Enumeration Theorem**
- ✓ Data can be effectively incorporated into a program and can effectively code programs as well – **Parametrization (Smn) Theorem**
- ✓ Kleene – **Universal Partial Recursive Function**, f(e,**x**) = g(**x**)
- ✓ Universal Turing Machine computes f
- ✓ Recursively enumerable relations – domain of a p.r.f
- ✓ *f(e) = e -* ***Quines*** *(Kleene's Fixpoint Theorem – such an 'e' exists)*

```
eval s="print 'eval s=';p s"
```

Curious Bob II…

- ✓ Wait a minute! So many definitions? – **They are all equivalent**
- ✓ How can "if there exists" be verified? – **No general way**
- ✓ Is there any way? – **Write programs! Hire coders! Google!**
- ✓ What if my code never stops running? – **:P**
- ✓ Can I check if this would happen? – **Sometimes**
- ✓ Come on! Sometimes won't suffice. In general? – **No**
- ✓ Who says that? – **Alan Turing**
- ✓ Is he the authority here? – **Yes, he is known as the Father of Computer Science**

The Halting Problem

Primary references :

[1] Floyd, R. W., "*Assigning meaning to programs*", American Math. Soc. 1967, pp. 19 – 32

[2] Aho, A. V., Ullman J. D., "*The theory of parsing, translation and compiling*", *Vol. 2, Prentice Hall, 1973*

[3] Katz, S., Manna, Z., "*A closer look at termination*", Acta Informatica 5, 1975, pp. 333 – 352

[4] Manna Z., Dershowitz, N., "*Proving termination with multiset orderings*", Memo AIM-310, Stanford Intelligence Laboratory, 1978

[5] Blass, A., Gurevich, Y., "*Proving termination and well-partial orderings*", ACM Transactions on computational logic, 2006, pp. 1 – 26

The actual problem …

- ✓ Optimized resource management and determinism of events
- ✓ Program must halt for a given set of inputs
- ✓ Existence of infinite computations in a program
- ✓ Generalized Halting Problem is algorithmically unsolvable – Undecidable
- ✓ Solving restricted instances

Known approaches…

- ✓ Floyd's approach – **No infinitely-descending-chain**
- ✓ Loop Approach
- ✓ Exit Approach
- ✓ Burstall's Approach
- ✓ Well-partial orderings
  - ✓ Multiset orderings

Floyd's Approach

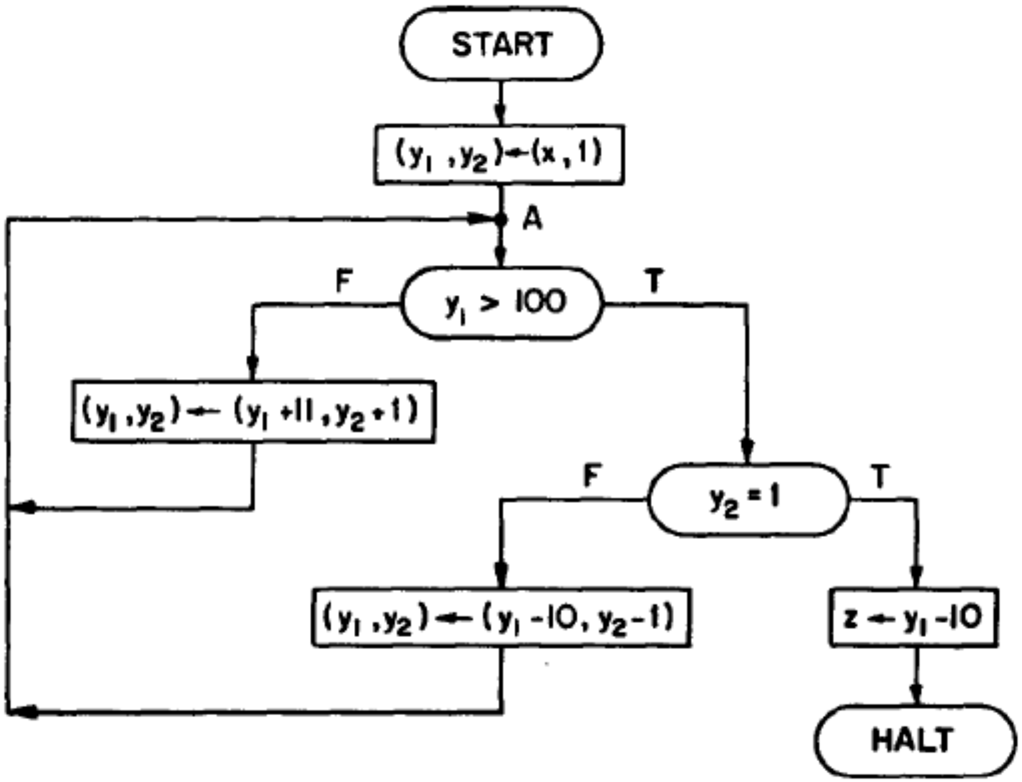


Fig. 1. The "91-function" program

$z =$ **if** $x > 101$ **then** $x - 10$ **else** 91

Loop Approach

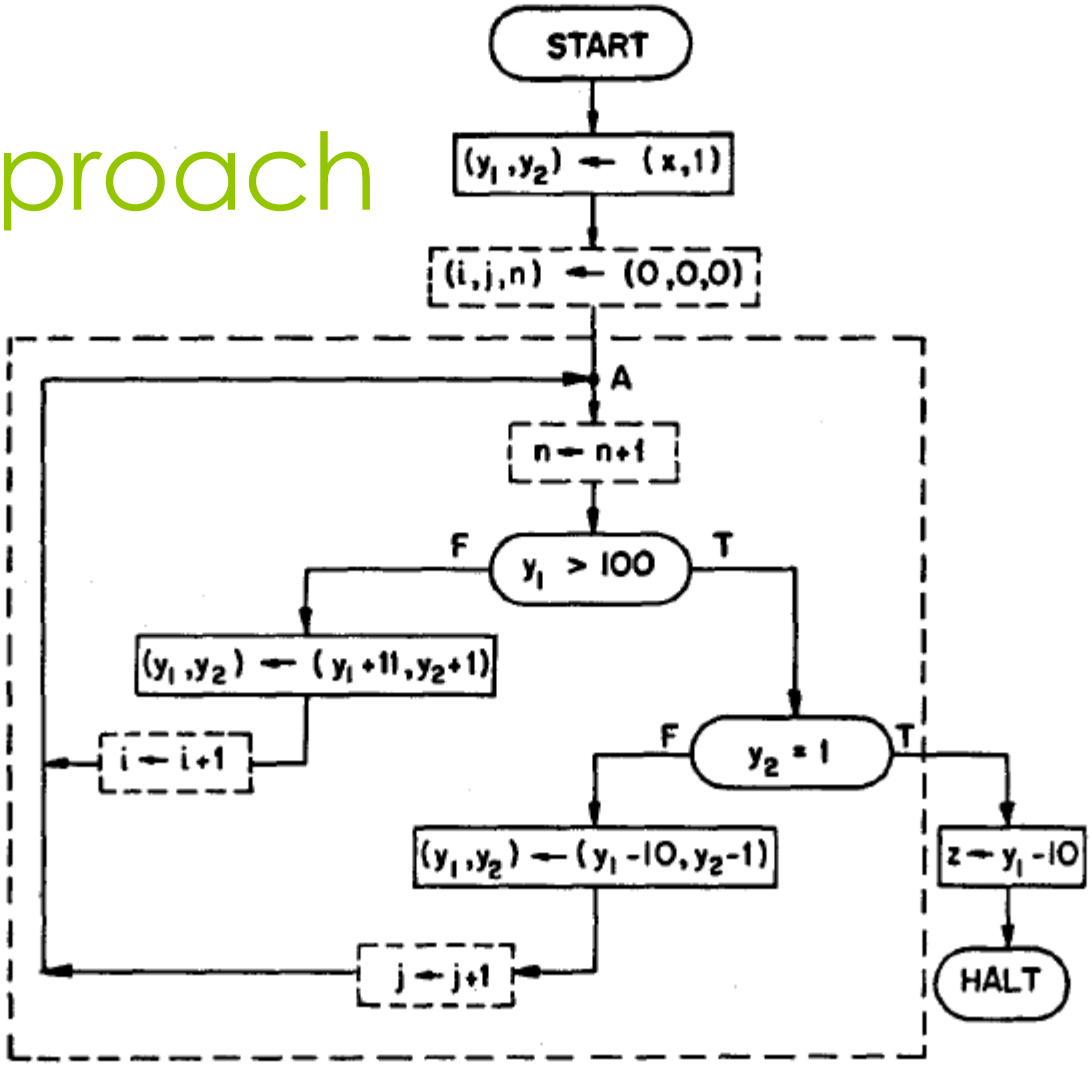


Fig. 4. The "91-function" program (with counters)

Exit Approach

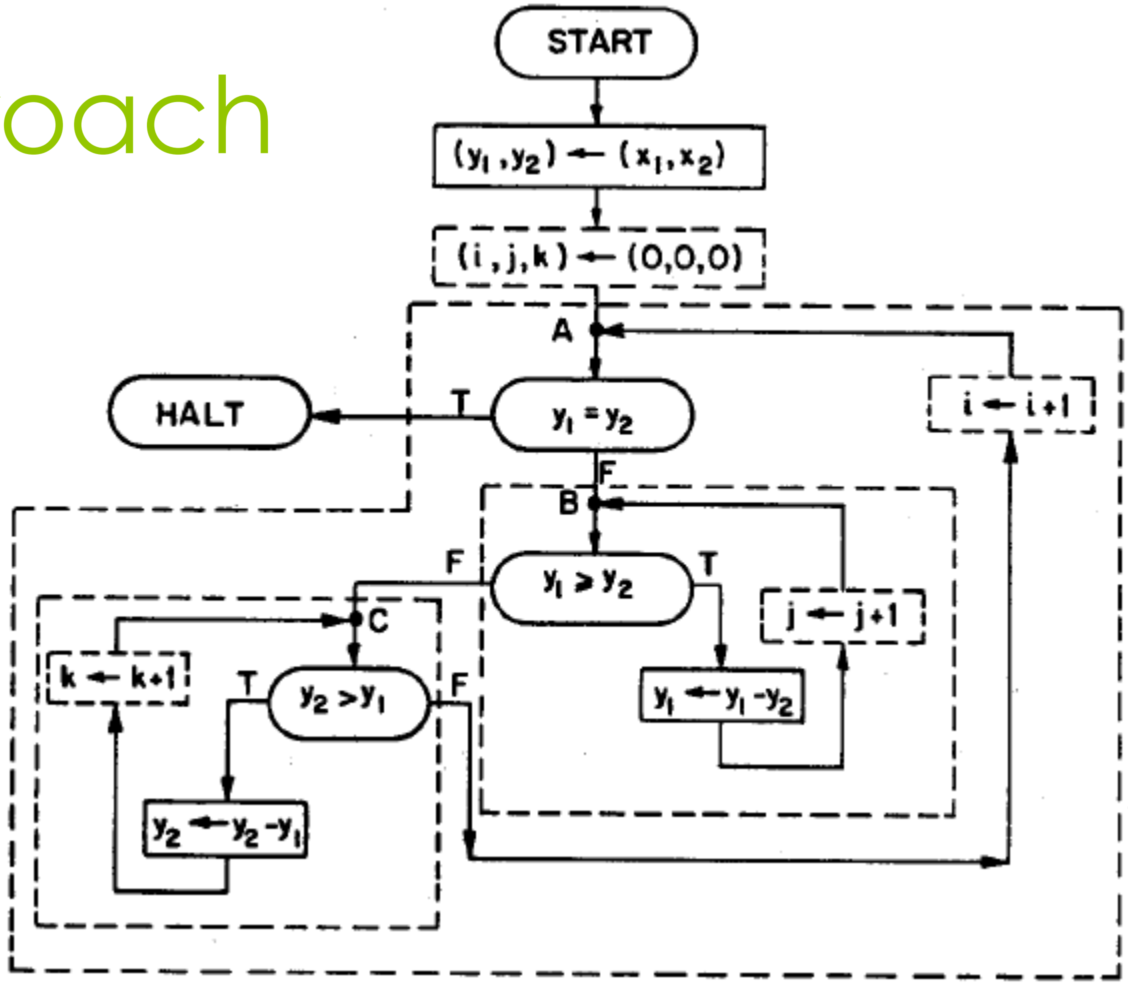


Fig. 8. A modified g.c.d. program

Structural Induction

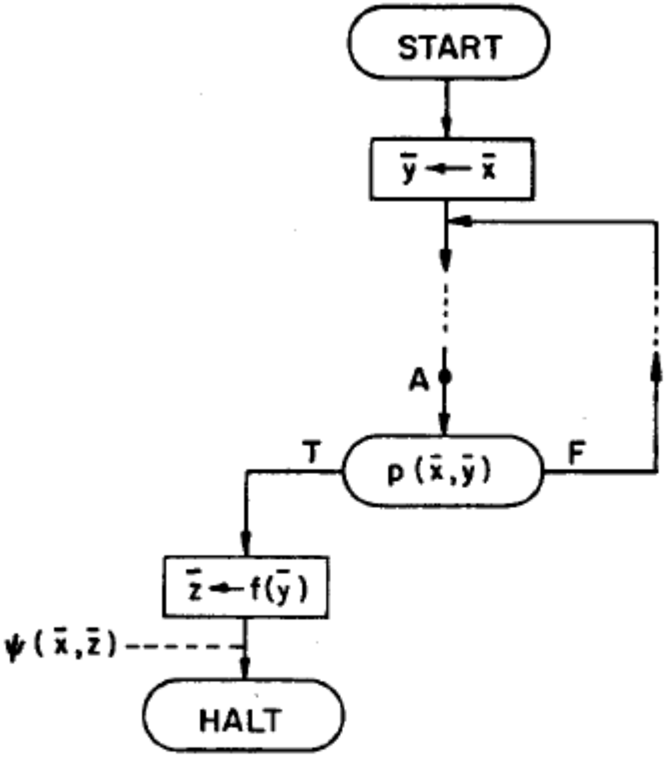


Fig. 9

Multiset Ordering …

```
Counting tips of Binary Trees
S := (t)
C := (0)
Loop until S = ()
     Y := head(S)
     if tip(y) then S := tail(S)
                    C := C+1
               Else S := left(s).right(S).tail(S)
               fi
Repeat
```

$$\tau(S) = \sum_{s \in S} nodes(S)$$

Using well-founded set ω

$$\tau(S) = \{s : s \in S\}$$

Using multiset theory

Thank you