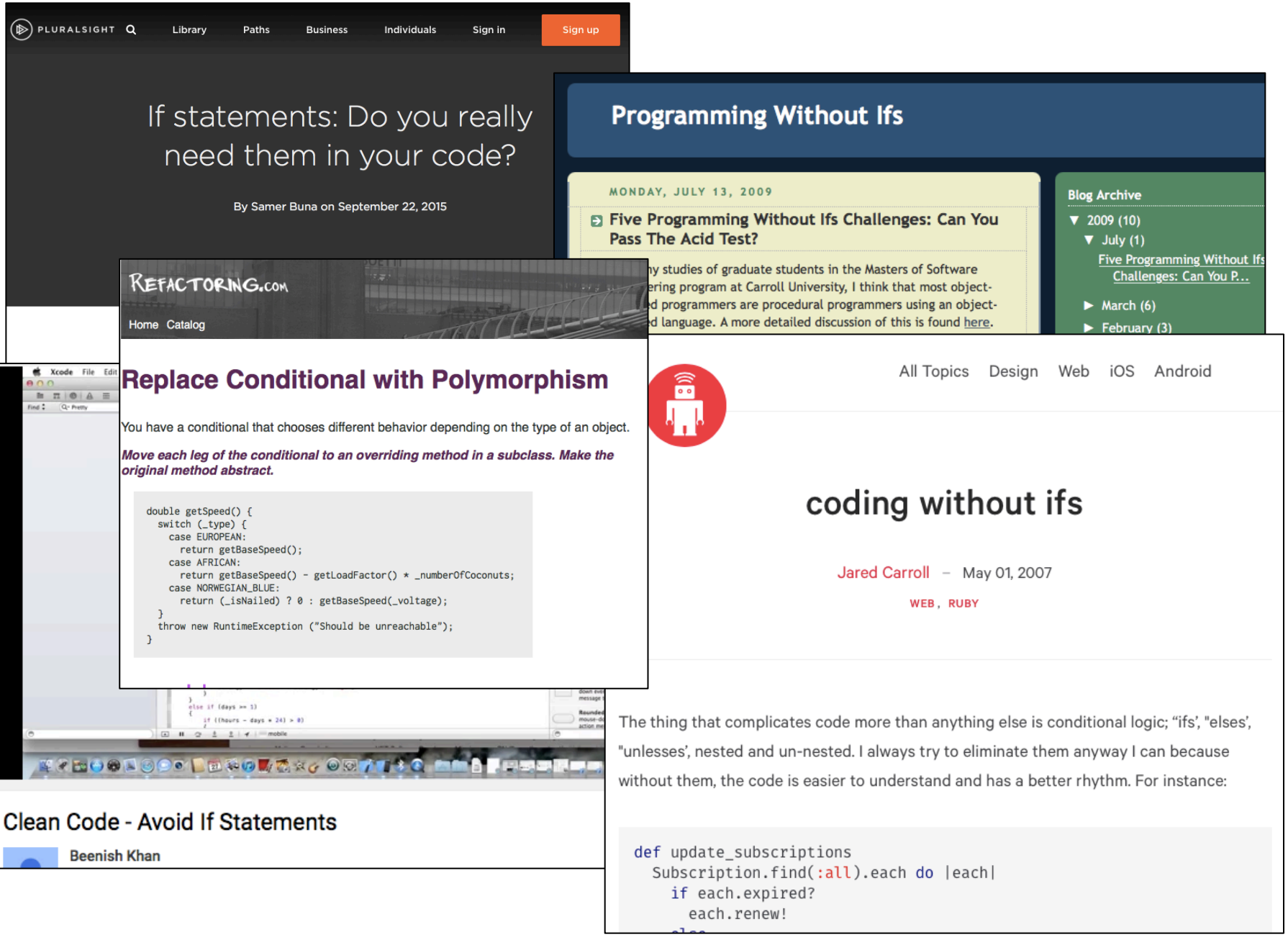
The Trouble With If

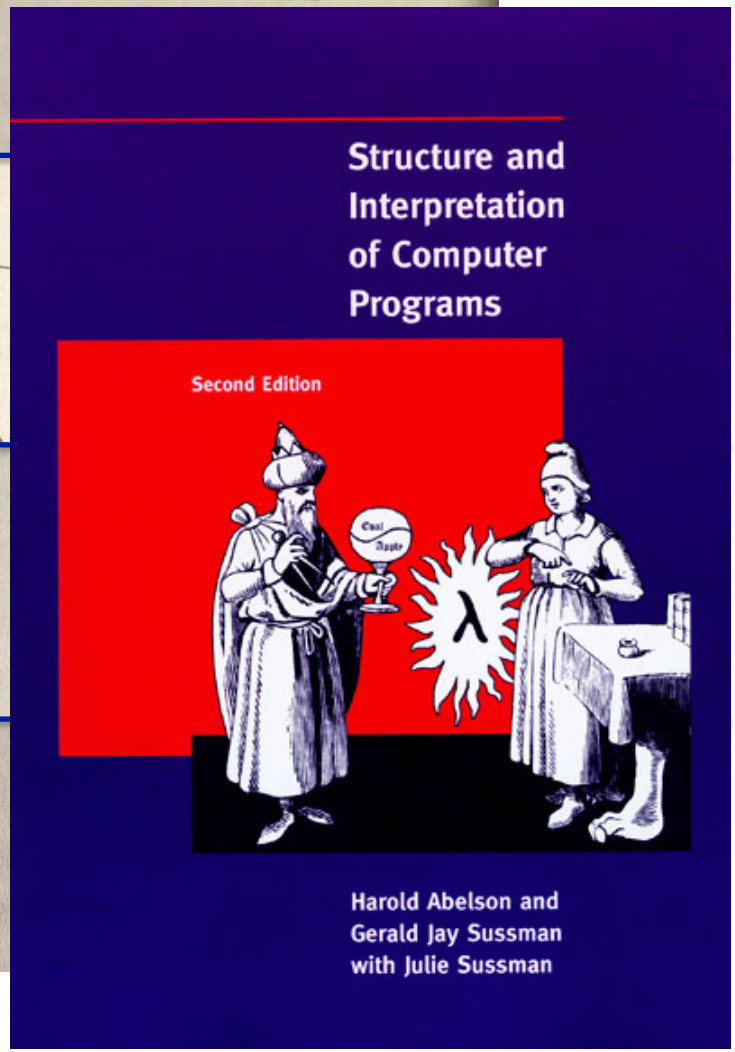
Eric Smith
@eric_s_smith

Vermont Functional Users Group
@vtfun

August 11, 2016


Michael Feathers
« The Single Responsibility and Open/Closed Prin
November 05, 2013
Unconditional Programming
Control structures have been around nearly as long
annoyance. Over and over again, I find that better
Often this happens because developers are using la
avoid control structures but they do.
If we are working in an object-oriented language, v
works well for if-statements too, but it can be over
we can do most of the work that we do in loops usi
and that can be a good thing.
The problem with control structures is that they of
statement:
if ...
...
else
...
end
Every place that we have ellipses in that code is a
outside of the if. It's very easy to introduce couplin
conditionals.
interspersed
You've
Death to the IF statement
November 8, 2013 code 115 comments edit
re and more interested in functional
pressiveness, and elegance they hold.
uage of choice and wander the desert
F#, Haskell, or Clojure to enjoy these
ming, Michael Feathers ponders how
nearly as long as programming but
Premise
Most ifs can be replaced by polymorphism
1:01 / 38:24
"The Clean Code Talks -- Inh
Polymorphism, & Testing"
GoogleTechTalks
Insights > Blog > Elimin
August 3, 2013
Eliminate Branching (I Statements) to Produce Better Code
Written By:
CHRIS WASH
ever devised. What
to write a program
Branching Consi
In 1968, E.W. Diskjs
be published in the
later renamed "Got
of its own in the CS community, denoting a practice that has fallen fr
ifs considered Harmful
Or; how to eliminate 90% of your bugs and 99% of your technical debt in one easy step.
Jules May
JulesMay.co.uk
0:05 / 1:26:46
If considered harmful: How to eradicate 95% of all your bugs in one simple step - Jules May


PLURALSIGHT
Library
Paths
Business
Individuals
Sign in
Sign up
If statements: Do you really need them in your code?
By Samer Buna on September 22, 2015
Programming Without Ifs
MONDAY, JULY 13, 2009
Five Programming Without Ifs Challenges: Can You Pass The Acid Test?
Blog Archive
2009 (10)
July (1)
Five Programming Without Ifs Challenges: Can You P...
March (6)
February (3)
REFACTORING.com
Home Catalog
Replace Conditional with Polymorphism
You have a conditional that chooses different behavior depending on the type of an object.
Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.
double getSpeed() {
switch (_type) {
case EUROPEAN:
return getBaseSpeed();
case AFRICAN:
return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
case NORWEGIAN_BLUE:
return (_isNailed) ? 0 : getBaseSpeed(_voltage);
}
throw new RuntimeException ("Should be unreachable");
}
All Topics
Design
Web
iOS
Android
coding without ifs
Jared Carroll – May 01, 2007
WEB, RUBY
The thing that complicates code more than anything else is conditional logic; “ifs’, "elses', "unlesses', nested and un-nested. I always try to eliminate them anyway I can because without them, the code is easier to understand and has a better rhythm. For instance:
def update_subscriptions
Subscription.find(:all).each do |each|
if each.expired?
each.renew!
Clean Code - Avoid If Statements
Beenish Khan



Abstraction barriers

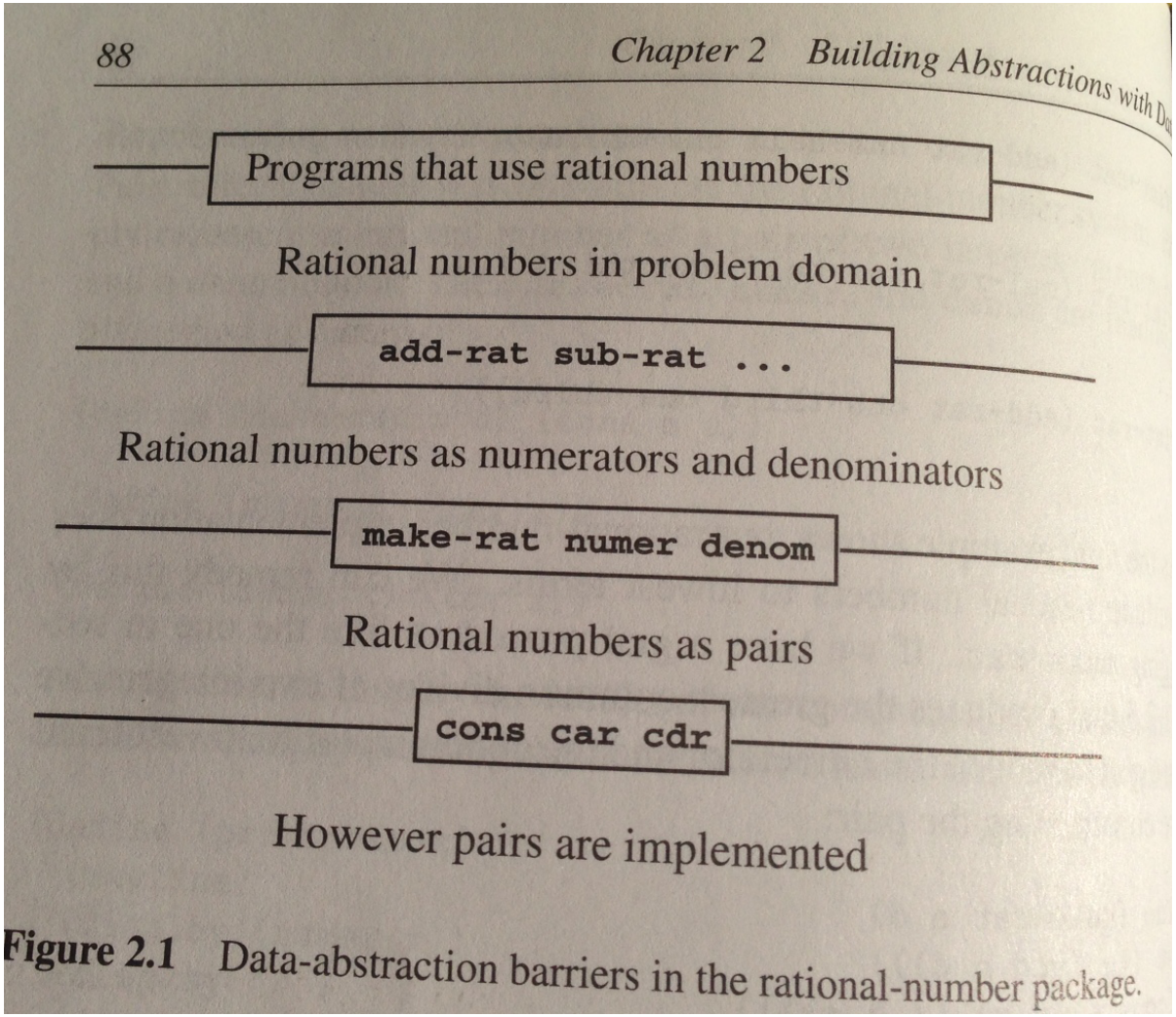


Figure 2.1 Data-abstraction barriers in the rational-number package.

Teaching creative computer science:
Simon Peyton Jones at TEDxExeter

"Computer Science has to do with the study of **information** and **computation**"

What's inside a program?

Whilst the detail will vary from one language to another, there are some common structures and ideas which programmers use over and over again from one language to another and from one problem to another:

- Sequence: running instructions in order (see below and page 25)
- Selection: running one set of instructions or another, depending on what happens (see pages 25–26)
- Repetition: running some instructions several times (see pages 26–27)
- Variables: a way of storing and retrieving data from the computer's memory (see pages 27–28).

Computational Linguistics

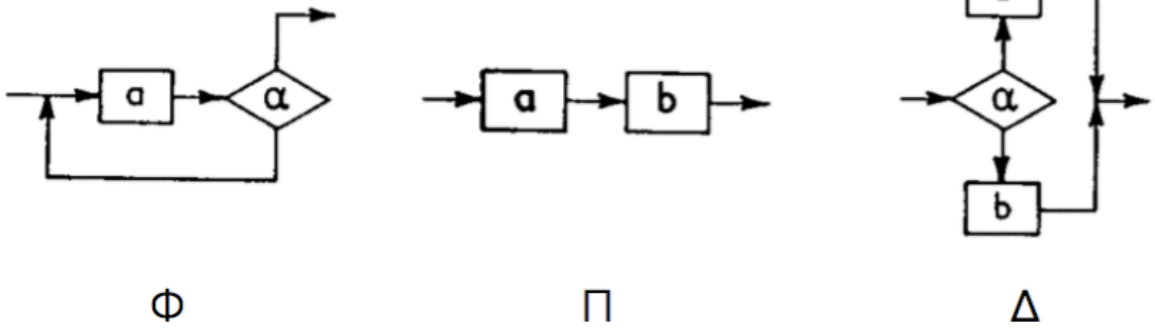
D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

CORRADO BÖHM AND GIUSEPPE JACOPINI
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite numbei of

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported



"It is thus proved possible to completely describe a program by means of a formula containing the names of diagrams Φ, Π, and Δ, "

Models of Computation

- Lambda calculus
- Turing machine
- RAM
- Register machine
- FSM
- Recursive functions
- Combinatory logic

Models of Computation

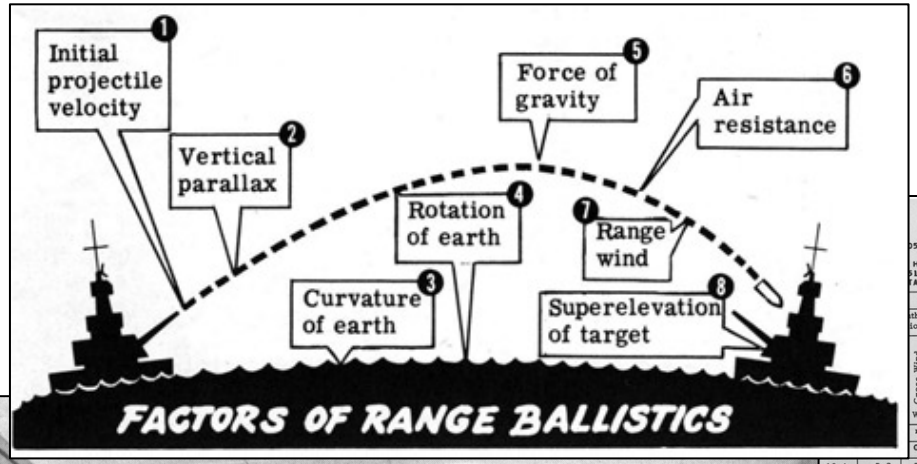
- Lambda calculus
- Turing machine
- RAM
- Register machine
- FSM
- Recursive functions
- Combinatory logic

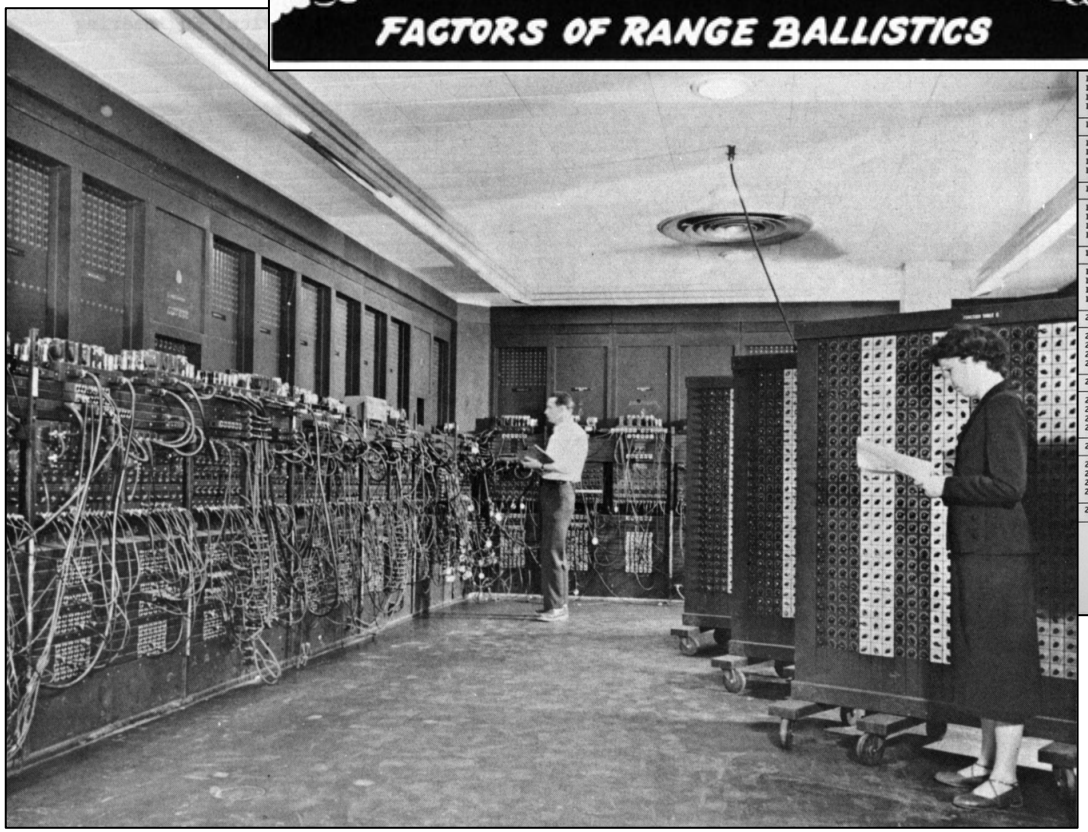
Church's *Thesis*

"We now define the notion, already discussed, of an effectively calculable function of positive integers by identifying it with the notion of a recursive function of positive integers (or of a lambda definable function of positive integers)."

- Alonzo Church (1936)

Random Access Machine





ENIAC

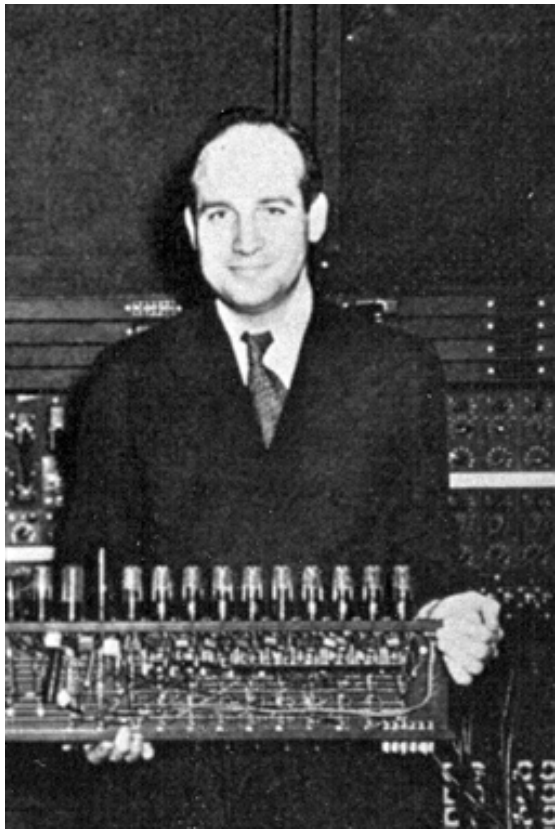
FT 105-AS-1 CORRECTION FACTORS 141

CARTRIDGE, HE, M1
FUZE, PD, M51 MODS
TABLE F

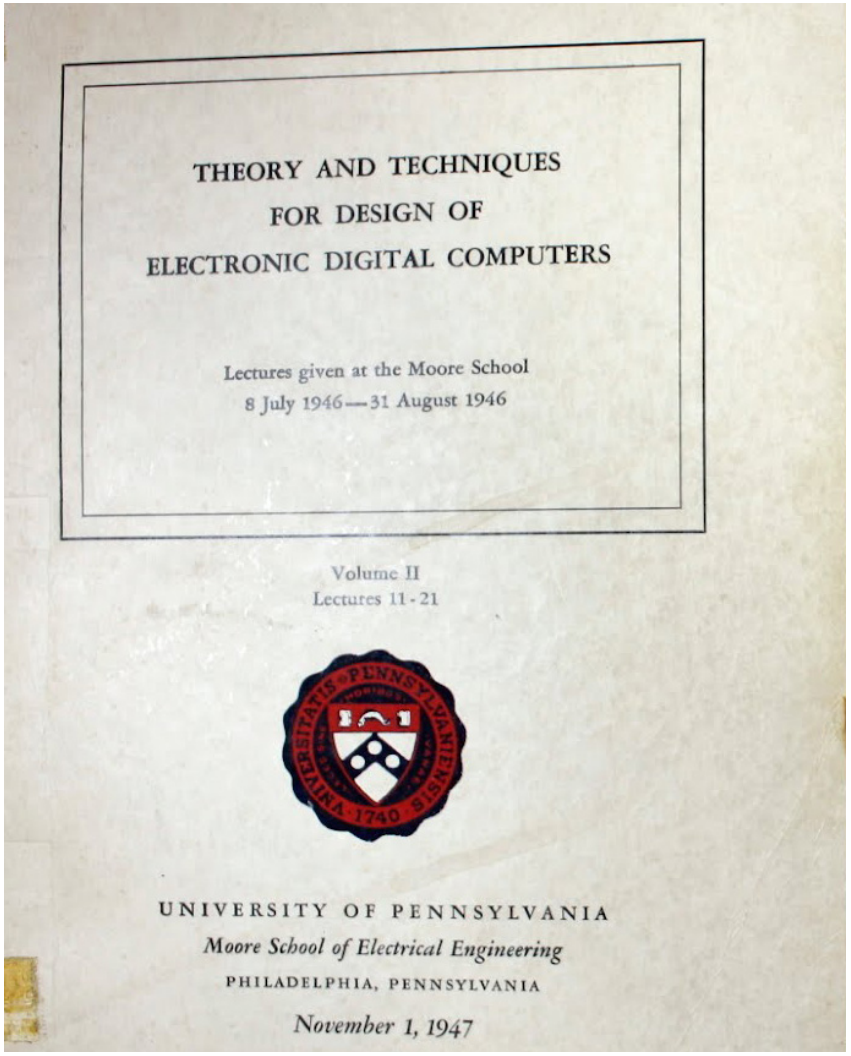
CHARGE 5

1	10	11	12	13	14	15	16	17	18	19
	Range Corrections For:									
	Muzzle Velocity 1 m/s		Range Wind 1 knot		Air Temp 1%		Air Pressure 1%		Proj wt of [illegible] is std	
Range	Decrease of	Increase of	Head Wind	Tail Wind	Decrease of	Increase of	Decrease of	Increase of	Decrease of	Increase of
R	M/S	M/S	W-R	W-R	Temp	Temp	Pres	Pres	Wt	Wt
m	m	m	m	m	m	m	m	m	m	m
3500	16.6	-11.1	5.6	-2.4	18.1	-8.7	-3.6	3.6	-20	23
3600	17.0	-11.3	5.8	-2.5	18.7	-9.0	-3.8	3.8	-20	24
3700	17.4	-11.6	6.0	-2.6	19.3	-9.4	-4.0	4.0	-21	24
3800	17.8	-11.8	6.2	-2.7	20.0	-9.7	-4.2	4.2	-21	24
3900	18.2	-12.1	6.4	-2.8	20.6	-10.1	-4.4	4.4	-21	25
4000	18.6	-12.3	6.6	-2.9	21.2	-10.4	-4.6	4.6	-21	25
4100	19.0	-12.6	6.8	-3.0	21.9	-10.8	-4.8	4.8	-22	25
4200	19.4	-12.8	7.0	-3.1	22.5	-11.1	-5.0	5.1	-22	26
4300	19.8	-13.1	7.2	-3.2	23.1	-11.5	-5.3	5.3	-22	26
4400	20.2	-13.3	7.4	-3.3	23.7	-11.8	-5.5	5.5	-22	26
4500	20.6	-13.6	7.6	-3.4	24.4	-12.2	-5.7	5.7	-23	27
4600	21.0	-13.8	7.8	-3.6	25.0	-12.5	-5.9	6.0	-23	27
4700	21.4	-14.1	8.0	-3.7	25.6	-12.9	-6.2	6.2	-23	27
4800	21.8	-14.4	8.2	-3.8	26.2	-13.3	-6.4	6.5	-23	28
4900	22.2	-14.6	8.4	-3.9	26.8	-13.6	-6.7	6.7	-23	28
5000	22.6	-14.9	8.6	-4.0	27.4	-14.0	-6.9	7.0	-24	28
5100	23.0	-15.1	8.8	-4.1	28.0	-14.3	-7.2	7.2	-24	28
5200	23.4	-15.4	9.0	-4.2	28.6	-14.7	-7.4	7.5	-24	29
5300	23.8	-15.6	9.2	-4.3	29.2	-15.0	-7.7	7.7	-24	29
5400	24.2	-15.9	9.4	-4.4	29.8	-15.4	-7.9	8.0	-24	29
5500	24.6	-16.2	9.6	-4.5	30.4	-15.7	-8.2	8.3	-24	29
5600	25.1	-16.4	9.8	-4.6	31.0	-16.1	-8.5	8.6	-24	30
5700	25.5	-16.7	9.9	-4.8	31.6	-16.4	-8.7	8.9	-25	30
5800	25.9	-16.9	10.1	-4.9	32.1	-16.7	-9.0	9.2	-25	30
5900	26.3	-17.2	10.3	-5.0	32.7	-17.1	-9.3	9.5	-25	30
6000	26.7	-17.5	10.5	-5.1	33.2	-17.4	-9.6	9.8	-25	31
6100	27.1	-17.7	10.6	-5.2	33.8	-17.8	-9.9	10.1	-25	31
6200	27.5	-18.0	10.8	-5.3	34.4	-18.1	-10.2	10.4	-25	31
6300	27.9	-18.3	11.0	-5.4	34.9	-18.5	-10.6	10.7	-25	31
6400	28.4	-18.6	11.1	-5.6	35.4	-18.8	-10.9	11.1	-25	31
6500	28.8	-18.9	11.3	-5.7	36.0	-19.1	-11.2	11.4	-26	32
6600	29.2	-19.2	11.5	-5.8	36.5	-19.5	-11.5	11.7	-26	32
6700	29.7	-19.4	11.6	-5.9	37.0	-19.8	-11.8	12.1	-26	32
6800	30.1	-19.7	11.8	-6.0	37.5	-20.1	-12.2	12.4	-26	32
6900	30.6	-20.0	11.9	-6.1	37.9	-20.4	-12.5	12.8	-26	32
7000	31.0	-20.3	12.0	-6.2	38.4	-20.8	-12.9	13.2	-26	33

Design of stored program computer



Presper Eckert

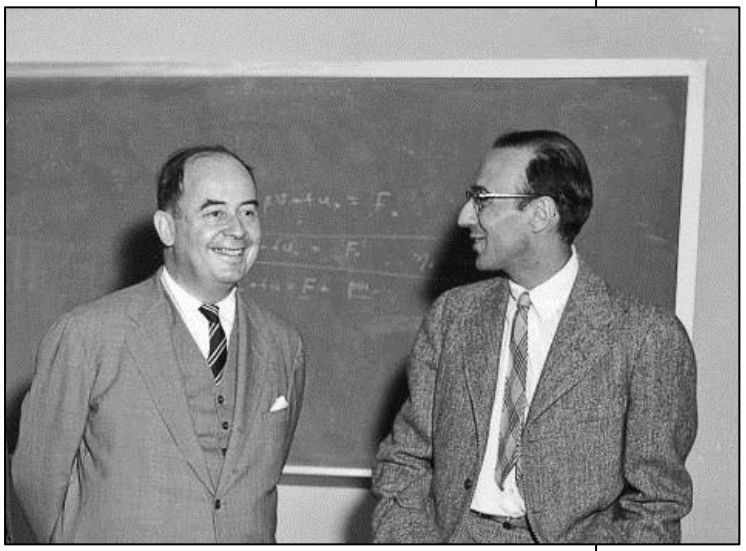


THEORY AND TECHNIQUES
FOR DESIGN OF
ELECTRONIC DIGITAL COMPUTERS

Lectures given at the Moore School
8 July 1946—31 August 1946

Volume II
Lectures 11-21

UNIVERSITY OF PENNSYLVANIA
Moore School of Electrical Engineering
PHILADELPHIA, PENNSYLVANIA
November 1, 1947



First Draft of a Report on the EDVAC

by

John von Neumann

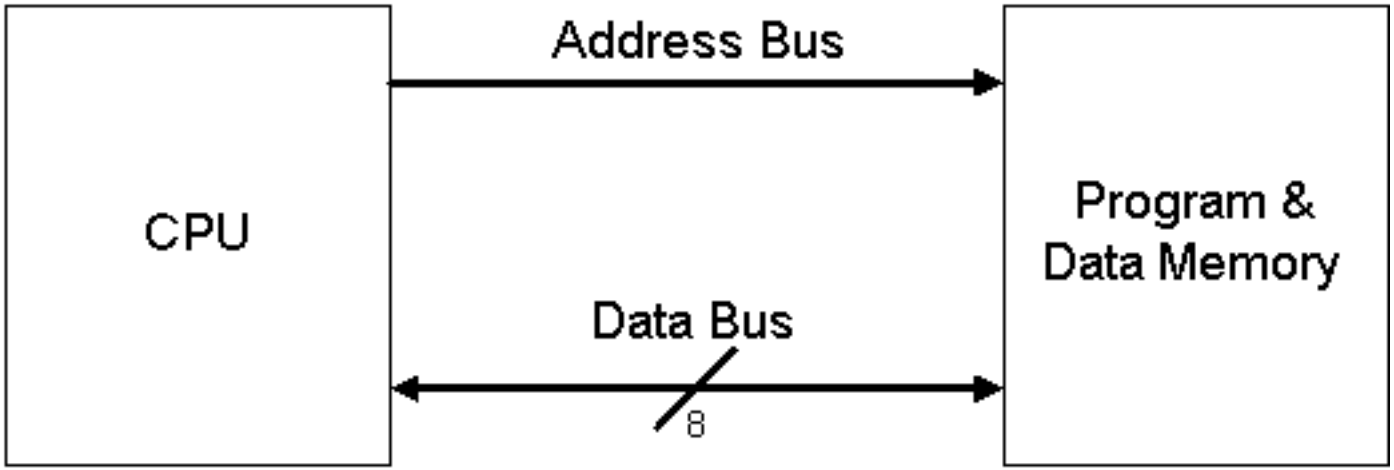
Contract No. W–670–ORD–4926

Between the

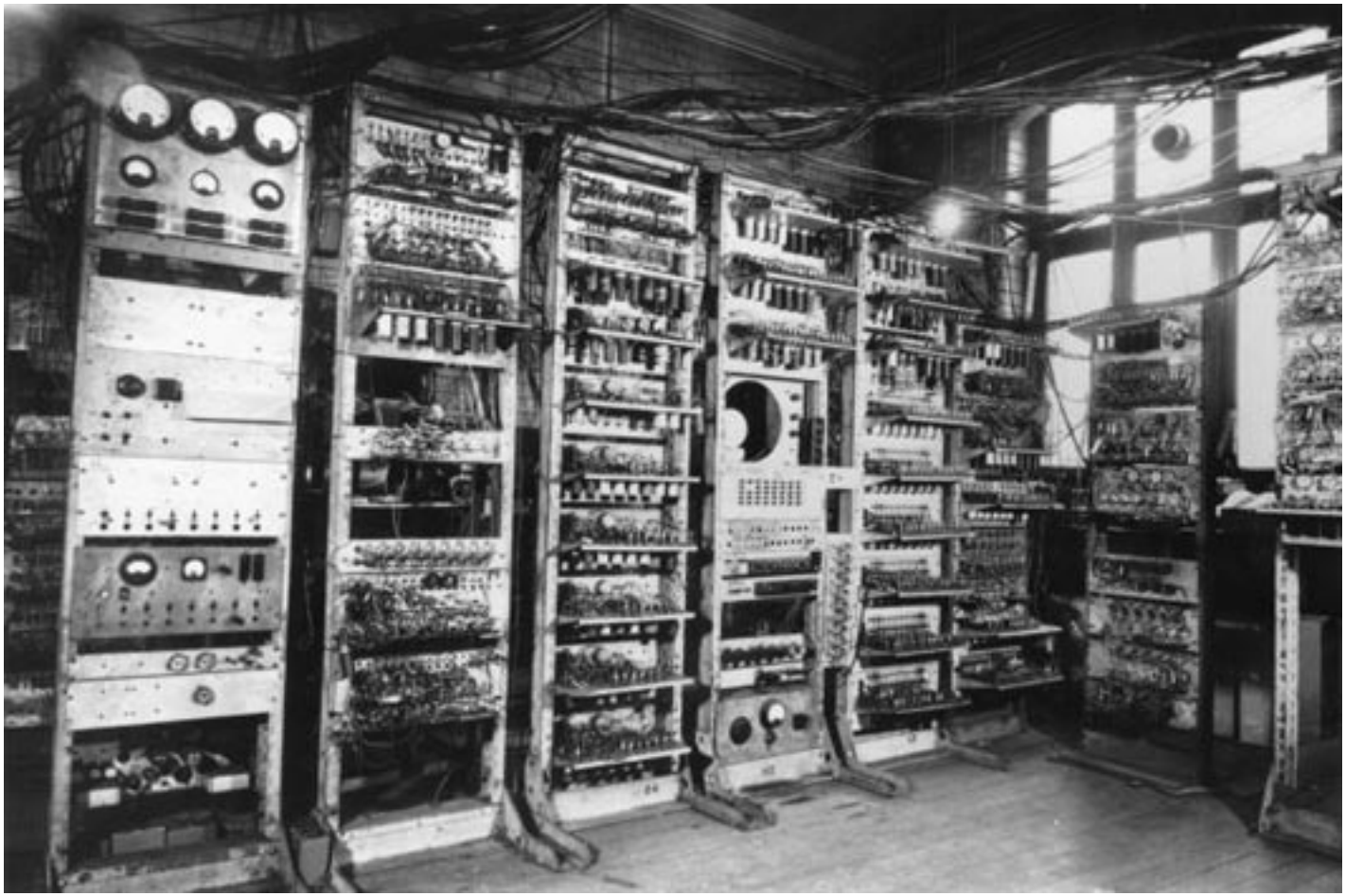
United States Army Ordnance Department

and the

University of Pennsylvania


CPU
Address Bus
Data Bus
8
Program &
Data Memory

June 21, 1948



University of Manchester SSEM

INC Ri	Increment contents of register Ri
DEC Ri	Decrement content of register Ri; Unchanged if 0
CLR Ri	Place 0 in register Ri
Ri <- Rj	Replace contents of register Ri with contents of Rj
JMP Nix	If x=a, execute instruction with label Ni above; x=b execute below
Rj JMP Nix	If register Rj contains 0, perform as above
CONTINUE	Do nothing

```
N1   R2 JMP N2b
     INC R1
     DEC R2
     JMP N1a
N2   Continue
```

A Recursive Introduction to the Theory of Computation, Smith 1994

We summarize this as follows:

TRA A A → C(ILC)

There are several conditional transfer instructions. Each of these has associated with it a condition which, if satisfied, causes the computer to take the next instruction from a specified storage location. If the condition is not satisfied, the computer takes the next instruction from the next storage location in normal sequence.

(2) Transfer on Minus

TMI A

causes the computer to take its next instruction from location A if the contents of the accumulator is negative and otherwise to execute the next instruction in normal sequence. We can summarize this as follows:

TMI A if (C(AC) negative) then A → C(ILC)

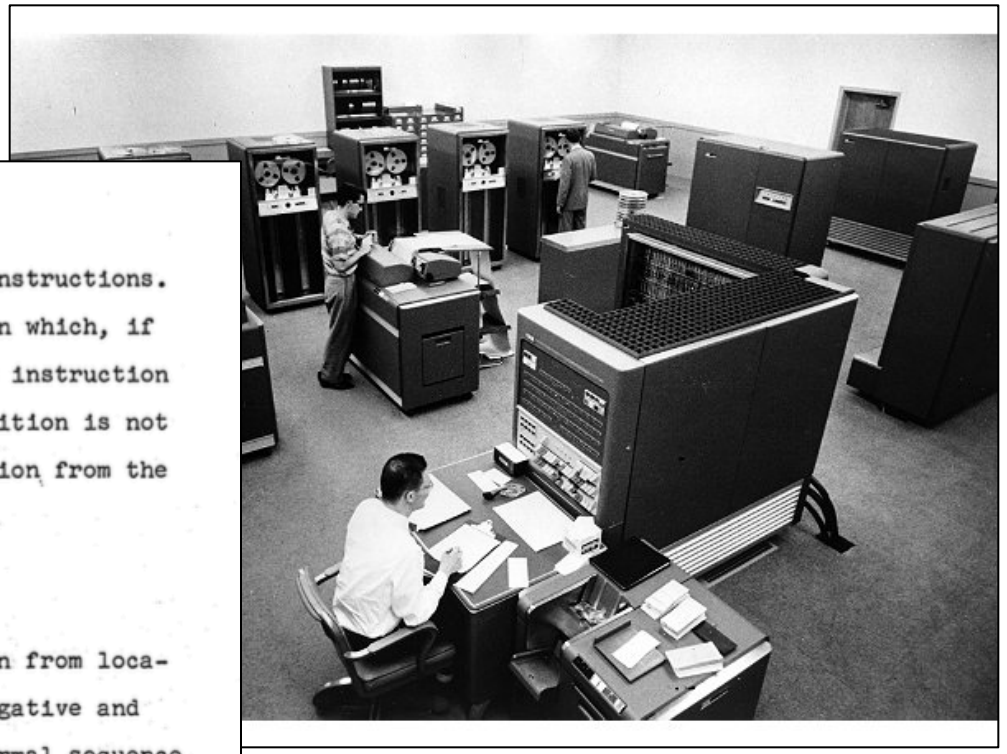
Some other conditional transfer instructions are:

(3) Transfer on Plus

TPL A if (C(AC) positive) then A → C(ILC)

(4) Transfer on MQ Plus

TQP A if (C(MQ) positive) then A → C(ILC)



Coding for MIT-IBM 704 Computer (1957)

REPERTOIRE OF MATH-MATIC CONTROL STATEMENTS

1. (A) READ△X△Z△.

   (B) READ△X△Z△IF△SENTINEL△JUMP△TO△SENTENCE

2. TYPE-IN△X△Z△.

3. PRINT-OUT△X△Z△.

4. (A) EDIT△X△Z△.

   (B) EDIT△CONVERTED△X△Z△.

   (C) EDIT△FOR△UNIPRINTER△X△Z△.

5. WRITE△X△Z△.

6. (A) EDIT△AND△WRITE△X△Z△.

   (B) EDIT△AND△WRITE△CONVERTED△X△Z△.

   (C) EDIT△CONVERTED△AND△WRITE△X△Z△.

   (D) EDIT△AND△WRITE△FOR△UNIPRINTER△X△Z△.

   (E) EDIT△FOR△UNIPRINTER△AND△WRITE△X△Z△.

   NOTE 1. IN THE ABOVE 5 SENTENCES EDIT A

7. (A) VARY△X△I₀Δ(Δ)△LX, SENTENCES△F△THRU△L△.

   (B) VARY△X△I₀Δ(Δ)△LX, Y△I₀Δ(Δ)△LY, SENTENCES△F△THRU△L△.

   (C) VARY△X△I₀Δ(Δ)△LX, Y△I₀Δ(Δ)△LY, Z△I₀Δ(Δ)△LZ, SENTENCES△F△THRU△L△.

8. (A) VARY△X△I₀ΔX_1△X_2△X_3△X_4Δ...△X_n△SENTENCES△F△THRU△L△.

   (B) VARY△X, Y, Z, X_0, Y_0, Z_0, X_1, Y_1, Z_1 ... X_n, Y_n, Z_n, SENTENCES△F△THRU△L△.

9. JUMP△TO△SENTENCE△F△.

10. (A) IF△X△r'△Y△JUMP△TO△SENTENCE△F△.

    (B) IF△X△r'△Y△JUMP△TO△SENTENCE△F_1, IF△
        X△r'△Y△JUMP△TO△SENTENCE△F_2, IF△
        X△r'△Y△JUMP△TO△SENTENCE△F_3△.

r' = Any of the three relations = equal to (=), greater than (>), less than (<).

Computed GO TO

GENERAL FORM	EXAMPLES
"GO TO (n_1, n_2, ..., n_m), i" where n_1, n_2, ..., n_m are statement numbers and i is a non-subscripted fixed point variable.	GO TO (30, 40, 50, 60), I

If at the time of execution the value of the variable i is j, then control is transferred to the statement with statement number n_j. Thus, in the example, if I has the value 3 at the time of execution, a transfer to statement 50 will occur. This statement is used to obtain a computed many-way fork.

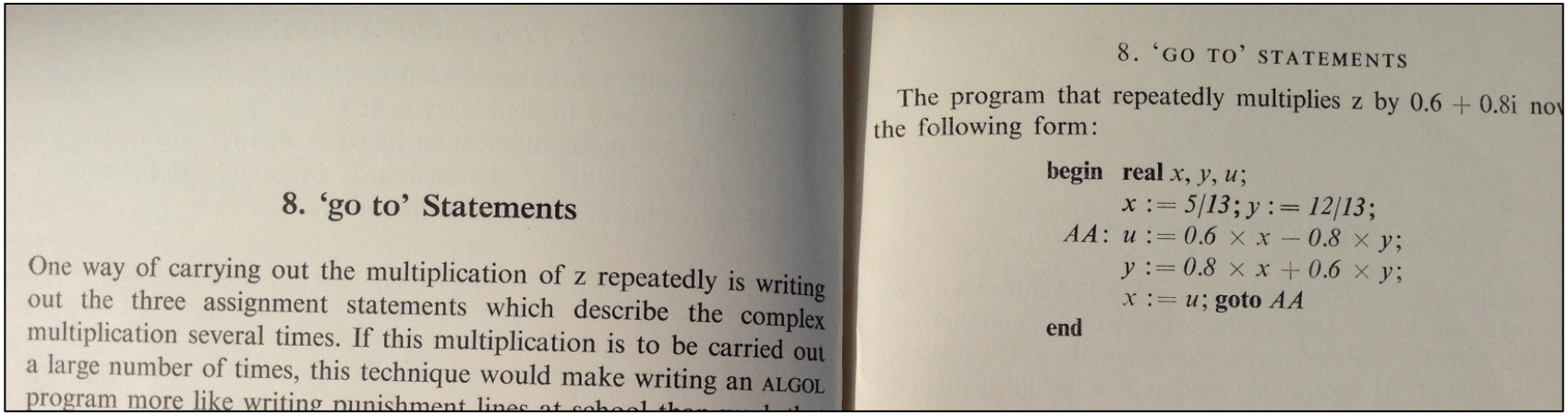
IF

GENERAL FORM	EXAMPLES
"IF (a) n_1, n_2, n_3" where a is any expression and n_1, n_2, n_3 are statement numbers.	IF (AU,K) —B) 10, 20, 30

Control is transferred to the statement with statement number n_1, n_2, or n_3 according as the value of a is less than, equal to, or greater than zero.

FORTRAN Automatic Coding System for IBM 704 (1956)

Preliminary Manual for MATH-MATIC and ARITH-MATIC Systems (1957)

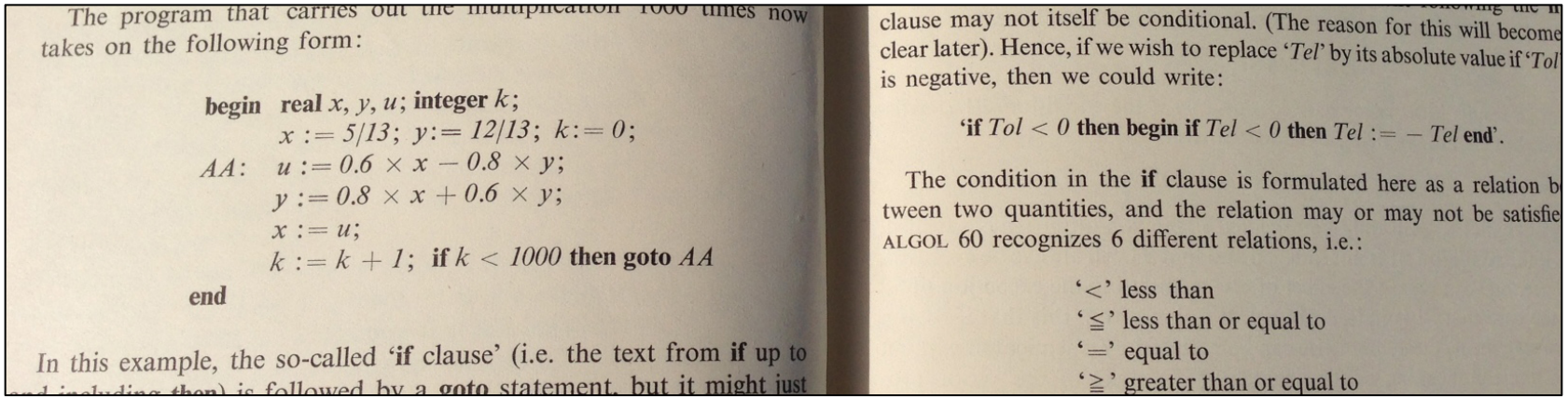


8. 'go to' Statements

One way of carrying out the multiplication of z repeatedly is writing out the three assignment statements which describe the complex multiplication several times. If this multiplication is to be carried out a large number of times, this technique would make writing an ALGOL program more like writing punishment lines at school

The program that repeatedly multiplies z by 0.6 + 0.8i nov the following form:

```
begin  real x, y, u;
       x := 5/13; y := 12/13;
AA:    u := 0.6 × x − 0.8 × y;
       y := 0.8 × x + 0.6 × y;
       x := u; goto AA
end
```



The program that carries out the multiplication 1000 times now takes on the following form:

```
begin  real x, y, u; integer k;
       x := 5/13; y := 12/13; k := 0;
AA:    u := 0.6 × x − 0.8 × y;
       y := 0.8 × x + 0.6 × y;
       x := u;
       k := k + 1; if k < 1000 then goto AA
end
```

In this example, the so-called 'if clause' (i.e. the text from **if** up to and including **then**) is followed by a **goto** statement, but it might just

clause may not itself be conditional. (The reason for this will become clear later). Hence, if we wish to replace '*Tel*' by its absolute value if '*Tol* is negative, then we could write:

'**if** *Tol* < *0* **then begin if** *Tel* < *0* **then** *Tel* := − *Tel* **end**'.

The condition in the **if** clause is formulated here as a relation b tween two quantities, and the relation may or may not be satisfie ALGOL 60 recognizes 6 different relations, i.e.:

'<' less than
'≦' less than or equal to
'=' equal to
'≧' greater than or equal to

Primer of Algol 60 Programming, Dijkstra (1962)

Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set

Top-down programming

This naturally suggests that programs should be developed by beginning at a high level of abstraction and repeatedly refining the level of detail. This approach, often called structured programming, top-down programming, or programming by stepwise refinement, has received considerable emphasis in recent years [Dijkstra 71, 72, Wirth 71b]. Although it is not a panacea, it is an immensely powerful tool for attacking complexity, and its employment in various guises will be a recurring theme throughout this book.

More precisely, we will say that a program is *structured* when it reveals a variety of levels of detail to the reader, and we will reserve the term *top-down* for the process of creating such a program by proceeding from the abstract to the concrete. Occasionally the opposite order of attack, which might be called *bottom-up* programming, is called for, particularly when the ultimate goal of the program is ill-defined or changeable.

From *The Craft of Programming,* p7 (1981)
John Reynolds

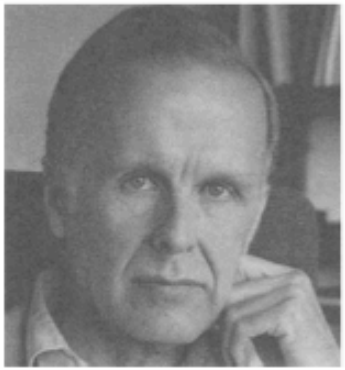
Blocks of spaghetti

"The spirit of block structure is a style, not a language. By simulating a Von Neumann machine, we can produce the behaviour of any spaghetti code within the confines of a block-structured language. This does not prevent it from being spaghetti."

- Bruce Mills, *Theoretical Introduction to Programming* (2006)

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic.

The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements of the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

Redacted

Page long method from real code base with nested ifs, extensive conditional logic, reassignments, dependence on earlier conditionals

A Complexity Measure

THOMAS J. McCABE

Abstract—This paper describes a graph-theoretic complexity measure and illustrates how it can be used to manage and control program complexity. The paper first explains how the graph-theory concepts apply and gives an intuitive explanation of the graph concepts in programming terms. The control graphs of several actual Fortran programs are then presented to illustrate the correlation between intuitive complexity and the graph-theoretic complexity. Several properties of the graph-theoretic complexity are then proved which show, for example, that complexity is independent of physical size (adding or subtracting functional statements leaves complexity unchanged) and complexity depends only on the decision structure of a program.

The issue of using nonstructured control flow is also discussed. A characterization of nonstructured control graphs is given and a method of measuring the "structuredness" of a program is developed. The relationship between structure and reducibility is illustrated with several examples.

The last section of this paper deals with a testing methodology used in conjunction with the complexity measure; a testing strategy is defined that dictates that a program can either admit of a certain minimal testing level or the program can be structurally reduced.

Index Terms—Basis, complexity measure, control flow, decomposition, graph theory, independence, linear, modularization, programming, reduction, software, testing.

I. Introduction

THERE is a critical question facing software engineering today: How to modularize a software system so the resulting modules are both testable and maintainable? That the issues of testability and maintainability are important is borne out by the fact that we often spend half of the development time in testing [2] and can spend most of our dollars maintaining systems [3]. What is needed is a mathematical technique that will provide a quantitative basis for

II. A Complexity Measure

In this section a mathematical technique for program modularization will be developed. A few definitions and theorems from graph theory will be needed, but several examples will be presented in order to illustrate the applications of the technique.

The complexity measure approach we will take is to measure and control the number of paths through a program. This approach, however, immediately raises the following nasty problem: "Any program with a backward branch potentially has an infinite number of paths." Although it is possible to define a set of algebraic expressions that give the total number of possible paths through a (structured) program,[1] using the total number of paths has been found to be impractical. Because of this the complexity measure developed here is defined in terms of basic paths—that when taken in combination will generate every possible path.

The following mathematical preliminaries will be needed, all of which can be found in Berge [1].

Definition 1: The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is

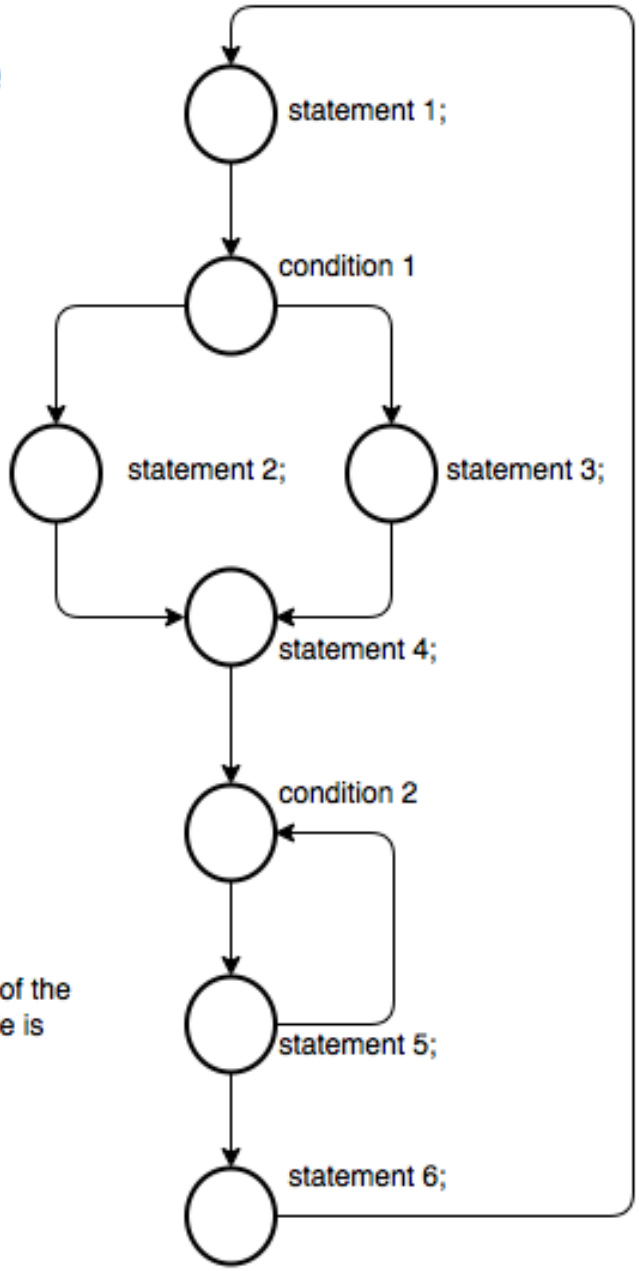
$$v(G) = e - n + p.$$

Theorem 1: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

The applications of the above theorem will be made as follows: Given a program we will associate with it a directed graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is classically known as the program

Sample code

```
statement 1;
If ( condition 1 ) {
        statement 2;
} else {
        statement 3;
}
statement 4;
for( condition 2 ) {
        statement 5;
}
statement 6;
```



Complexity

The cyclomatic complexity of the graph representing the code is

v(G) = E - N + 2

= 9 - 8 + 2

= 3

“The withering away of the statement” is a phenomenon we may live to see.

- Peter Landin, *Getting Rid of Labels*

Take-aways

- The stored program computer design (EDVAC) was spread through the Moore School lectures
- Instructions and data were treated uniformly by being stored in the same way.
- 70 years later, it is still how computers are designed.
- There is nothing particularly mathematical about it.
- Statements in programming languages emulate the hardware design.
- Statements encourage complexity.

Recursive Functions

Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I[1].

Von **Kurt Gödel** in Wien.

1.

Die Entwicklung der Mathematik in der Richtung zu größerer Exaktheit hat bekanntlich dazu geführt, daß weite [illegible] formalisiert wurden, in der Art, daß das Beweisen wenigen mechanischen Regeln vollzogen werden kan sendsten derzeit aufgestellten formalen Systeme sind Principia Mathematica (PM)[2]) einerseits, das Zerm sche (von J. v. Neumann weiter ausgebildete) Axio Mengenlehre[3]) andererseits. Diese beiden Systeme si alle heute in der Mathematik angewendeten Beweismet formalisiert, d. h. auf einige wenige Axiome und Schlu geführt sind. Es liegt daher die Vermutung nahe, da und Schlußregeln dazu ausreichen, alle mathematisch sich in den betreffenden Systemen überhaupt form lassen, auch zu entscheiden. Im folgenden wird ge nicht der Fall ist, sondern daß es in den beid Systemen sogar relativ einfache Probleme aus der Theorie der gewöhnlichen ganzen Zahlen gibt[4]), die sich aus den Axiomen nicht

On Formally Undecidable Propositions of Principia Mathematica and Related Systems

„0“ . . . 1 „∨“ . . . 7 „(“ . . . 11

„f“ . . . 3 „Π“ . . . 9 „)“ . . . 13

„∼“ . . . 5

ferner den Variablen n-ten Typs die Zahlen de
eine Primzahl > 13 ist). Dadurch entspricht jed
von Grundzeichen (also auch jeder Formel) in ei
eine endliche Reihe natürlicher Zahlen. Die endli
licher Zahlen bilden wir nun (wieder eineindeut
Zahlen ab, indem wir der Reihe $n_1, n_2, \ldots n_k$ die Z
entsprechen lassen, wo p_k die k-te Primzahl (de
deutet. Dadurch ist nicht nur jedem Grundzeich
jeder endlichen Reihe von solchen in eineinde
natürliche Zahl zugeordnet. Die dem Grundzeichen
zeichenreihe) a zugeordnete Zahl bezeichnen wir
irgend eine Klasse oder Relation $R(a_1, a_2 \ldots a_n)$

dnen ihr diejenige
rlichen Zahlen zu,
x_n besteht, wenn
$= 1, 2, \ldots n$) und
ationen natürlicher

Gödel Numbering

0	1	∨	7	(	11
succ	3	∀	9	)	13
¬	5				

type one variables x_1, y_1, z_1
type two variables x_2, y_2, z_3

¬ (x_2 (succ y_1))

$2^5 3^{11} 5^{289} 7^{11} 11^3 13^{17} 17^{13} 19^{13}$ = really big, unique number

unentscheidbare Probleme gibt, lautet z. B. folgendermaßen: Es gibt *Satzformeln* a, so daß weder a noch die *Negation* von a *beweisbare Formeln* sind.

Wir schalten nun eine Zwischenbetrachtung ein, die mit dem formalen System P vorderhand nichts zu tun hat, und geben zunächst folgende Definition: Eine zahlentheoretische Funktion[25]) $\varphi(x_1, x_2 \ldots x_n)$ heißt rekursiv definiert aus den zahlentheoretischen Funktionen $\psi(x_1, x_2 \ldots x_{n-1})$ und $\mu(x_1, x_2 \ldots x_{n+1})$, wenn für alle $x_2 \ldots x_n$, k[26]) folgendes gilt:

$$\begin{aligned} &\varphi(0, x_2 \ldots x_n) = \psi(x_2 \ldots x_n) \\ &\varphi(k+1, x_2 \ldots x_n) = \mu(k, \varphi(k, x_2 \ldots x_n), x_2 \ldots x_n). \end{aligned} \tag{2}$$

Eine zahlentheoretische Funktion φ heißt rekursiv, wenn es eine endliche Reihe von zahlentheor. Funktionen $\varphi_1, \varphi_2 \ldots \varphi_n$ gibt, welche mit φ endet und die Eigenschaft hat, daß jede Funktion φ_k der Reihe entweder aus zwei der vorhergehenden rekursiv definiert ist oder

aus irgend welchen der vorhergehenden durch Einsetzung entsteht[27]) oder schließlich eine Konstante oder die Nachfolgerfunktion $x+1$ ist. Die Länge der kürzesten Reihe von φ_i, welche zu einer rekursiven Funktion φ gehört, heißt ihre Stufe. Eine Relation zwischen natürlichen Zahlen $R(x_1 \ldots x_n)$ heißt rekursiv[28]), wenn es eine rekursive Funktion $\varphi(x_1 \ldots x_n)$ gibt, so daß für alle $x_1, x_2 \ldots x_n$

$$R(x_1 \ldots x_n) \sim [\varphi(x_1 \ldots x_n) = 0][29]).$$

Es gelten folgende Sätze:

General recursive functions of natural numbers[1]).

Von

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

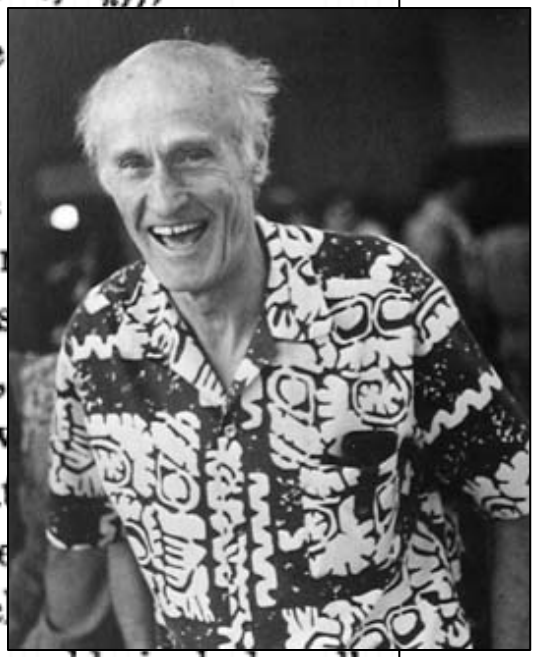
The substitution

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

where $\theta, \chi_1, \ldots, \chi_m, \psi, \chi$ are given functions of natu
examples of the definition of a function φ by equations
step by step process for computing the value $\varphi(k_1,$
given set $k_1, \ldots, k_n$ of natural numbers. It is know
other definitions of this sort, e. g. certain recursions wi
or more variables simultaneously, which cannot be reduce
of substitutions and ordinary recursions[2]). Hence, a c
the notion of recursive definition in general, which would include all



AN ALGEBRAIC LANGUAGE FOR THE MANIPULATION OF SYMBOLIC EXPRESSIONS

by John McCarthy

Abstract: This memorandum is an outline of the specification of an incomplete algebraic language for manipulating symbolic expressions. The
pleteness lies in the fact that while I am c
that the language so far developed and descr
is adequate and even more convenient than an
vious language for describing symbolic manip
certain details of the process have to be ex
mentioned in some cases and can be left to t
in others. This memorandum is only an outli
is sketchy on some important points.

LETTERS TO THE EDITOR

Dear Editor:

The object of this note is to advocate that the IAL[1] language be extended to include two additional notations: *conditional expressions* and *recursive definitions*. We shall describe the notations first and then argue for their inclusion in the language.

1. *Conditional Expressions*

IAL as presently proposed already allows propositional or Boolean expressions which have truth values (T or F) as values. Examples of such expressions are

$$X < Y$$
$$((X < 3) \vee (y > Z)) \wedge P$$

In the last example X and Y are numerical quantities while P is a propositional quantity.

The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence of truth values on other truth values by logical connectives. However, the notations for expressing symbolically the dependence of quantities of other kinds on truth values is inadequate, so that English words and phrases are generally used for expressing these dependences in texts that describe other dependences symbolically. For example, the function $|x|$ is usually defined in words.

Conditional expressions are a device for expres
dependence of quantities on propositional quant
conditional expression has the form

$$(p_1 \to e_1\ ,\ \cdots\ ,\ p_n \to e_n)$$

where the p's are propositional expressions and
are expressions of any kind. It may be read, "If
e_1, otherwise if p_2 then e_2, $\cdots$, otherwise if

expressions are undefined, a property we shall need later. If none of the p's in a conditional expression are true, its value is undefined and the program should go to an error routine. Some additional examples will illustrate the above point.

$$(2 < 1 \to 0/0,\quad T \to 3) = 3.$$
$$(2 < 1 \to 3,\quad T \to 0/0) \text{ is undefined.}$$
$$(2 < 1 \to 3,\quad 4 < 1 \to 4) \text{ is undefined.}$$

Some standard useful functions may be defined conveniently as conditional expressions.

$$|x| = (x < 0 \to -x,\quad T \to x)$$
$$\delta_{ij} = (i = j \to 1,\quad T \to 0)$$
$$\operatorname{sgn}(x) = (x < 0 \to -1,\quad x = 0 \to 0,\quad T \to 1)$$

2. *Recursive Function Definitions*

By using conditional expressions we can, without circularity, define functions by formulas in which the defined function occurs. For example, we write

$$n! = (n = 0 \to 1,\quad T \to n \cdot (n-1)!)$$

When we use this formula to evaluate $0!$ we get the answer 1; because of the way in which the value of a conditional expression was defined, the meaningless expression $0 \cdot (0-1)!$ does not arise. The evaluation of $2!$ according to this definition proceeds as follows:

$$\gcd(m, n) = (m > n \to \gcd(n, m),\quad \operatorname{rem}(n, m) = 0 \to m,\quad T \to \gcd(\operatorname{rem}(n, m), m)),$$

where $\operatorname{rem}(n, m)$ denotes the remainder left when n is divided by m.

factory, both practically and esthetically.

In the languages used for digital computer programming, we find much more adequate provisions for selection of different computation "branches." But practical computer languages do not lend themselves to formal mathematical treatment—they are not designed to make it easy to prove theorems about the procedures they describe. In a paper by McCarthy [1963], we find a formalism that enhances the practical aspect of the recursive-function concept, while preserving and improving its mathematical clarity.

McCarthy introduces "conditional expressions" of the form

$$f = (\textbf{if } p_1 \textbf{ then } e_1 \textbf{ else } e_2)$$

193

tement (or equation) that may be

lue of f is e_1. If not,

places the artificial multiplication trick. It also does more; it has also the power of the minimization operator. In fact it can give us the recursive function $T(n)$ directly from the Turing transformation equations of 10.1:

$$T(q, s, m, n) = (\textbf{if } q = 1 \textbf{ then } n \textbf{ else } T(q^*, s^*, m^*, n^*))$$

Note that this eliminates entirely the parameter t which carries the information about the length of the computation, as well as eliminating the minimization operator. The McCarthy formalism is like the general recursive (Kleene) system, in being based on some basic functions, composition, and equality, but with the conditional expression alone replacing both the primitive-recursive scheme and the minimization operator.

It is different, however, in that it has explicit provisions for self-reference, so that instead of using tricky arithmetic methods it can describe enumerations, universal processes, etc. more

Computation: Finite and Infinite Machines, Minsky (1967)

RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS*

J. McCARTHY

JOHN MCCARTHY, *Massachusetts Institute of Technology, Cambridge, Mass.*

1. Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

In this article, we first describe a formalism for defining functions recursively. We believe this formalism has advantages both as a programming language and as vehicle for developing a theory of computation. Next, we describe S-expressions and S-functions, give some examples, and then describe the universal S-function *apply* which plays the theoretical role of a universal Turing machine and the practical role of an interpreter. Then we describe the

2. Functions and Function Definitions

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known, but the notion of *conditional expression* is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

a. *Partial Functions.* A partial function is a function that is defined only on part of its domain. Partial functions necessarily arise when functions are defined by computations because for some values of the arguments the computation defining the value of the function may not terminate. However, some of our elementary functions will be defined as partial functions.

b. *Propositional Expressions and Predicates.* A propositional expression is an expression whose possible values are T (for truth) and F (for falsity). We shall assume that the reader is familiar with the propositional connectives $\wedge$ ("and"), $\vee$ ("or"), and $\sim$ ("not"). Typical propositional expressions are:

$$x < y$$

$$(x < y) \wedge (b = c)$$

$$x \text{ is prime}$$

A predicate is a function whose range consists of the truth values T and F.

c. *Conditional Expressions.* The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence

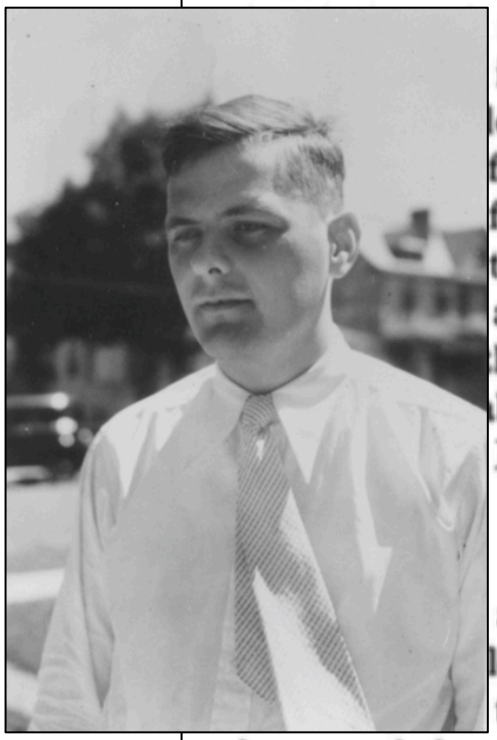
Take-aways

- Gödel published the first description of a computational model.
- Gödel's numbering demonstrated encoding of syntax patterns.
- McCarthy reconceived programming language as being built on the idea of recursion, instead of abstracting the hardware. This gave his language (LISP) mathematical properties.
- Conditional expressions allowed him to implement his recursive language on top of the existing computing machines.
- Programs and data were treated uniformly as symbolic expressions.

Lambda Calculus

A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.[1]

By Alonzo Church.[2]



1. **Introduction.** In this paper we present a set of postulates for the formal logic, in which we avoid use of the free, or real, in which we introduce a certain restriction on the law of le as a means of avoiding the paradoxes connected with the f the transfinite.

for avoiding use of the free variable is that we require that tion of symbols belonging to our system, if it represents at all, shall represent a particular proposition, unambigou- hout the addition of verbal explanations. That the use of ble involves violation of this requirement, we believe is For example, the identity

$$a(b+c) = ab+ac$$

and c are used as free variables, does note state a definite less it is known what values may be taken on by these this information, if not implied in the context, must be given by a verbal addition. The range allowed to the variables a, b, and c

ture, and in developing this formal structure reference to the proposed application must be held irrelevant. There may, indeed, be other applications of the system than its use as a logic.

translated into symbolic language, and, in order to make the translation

$$L, M, N \quad ::= \quad x$$
$$\mid \quad (\lambda x.\, N)$$
$$\mid \quad (L\, M)$$

α-conversion

```
λX.[...X...] = λY.[...Y...]
```

β-conversion

```
(λX.[...X...])(T) = [...T...]
```

η-conversion

```
λX.F(X) = F
```

Abstraction

$$f(x) = x^2 + 4x + 5$$

$$f = \lambda x.\ x^2 + 4x + 5$$

Application

$$(\lambda x.\ x^2 + 4x + 5)\ (y+3)$$

$$(y+3)^2 + 4(y+3) + 5$$

$f(y+3)$ where $f(x) = x^2 + 4x + 5$

S. GORN, Editor;

A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I*

By P. J. Landin†

This pa… LGOL
60 can … dence
between … in a
modified … com-
puter lang… ed on
the notion… action,
but also h… atures.
Then this … " into
which ALG… atures

The mechanical evaluation of expressions

By P. J. Landin

This paper is a contribution to the "theory" of the activity of using computers. It shows how some forms of expression used in current programming languages can be modelled in Church's λ-notation, and then describes a way of "interpreting" such expressions. This suggests a method, of analyzing the things computer users write, that applies to many different problem orientations and to different phases of the activity of using a computer. Also a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations.

Introduction

The point of departure of this paper is the idea of a machine for evaluating schoolroom sums, such as

1. $(3 + 4)(5 + 6)(7 + 8)$
2. if $2^{19} < 3^{12}$ then $\sqrt[12]{2}$ else $\sqrt[53]{2}$

is written explicitly and prefixed to its operand(s), and each operand (or operand-list) is enclosed in brackets, e.g.

$$f(a, +(\times(2, b), 3)).$$

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a

differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When

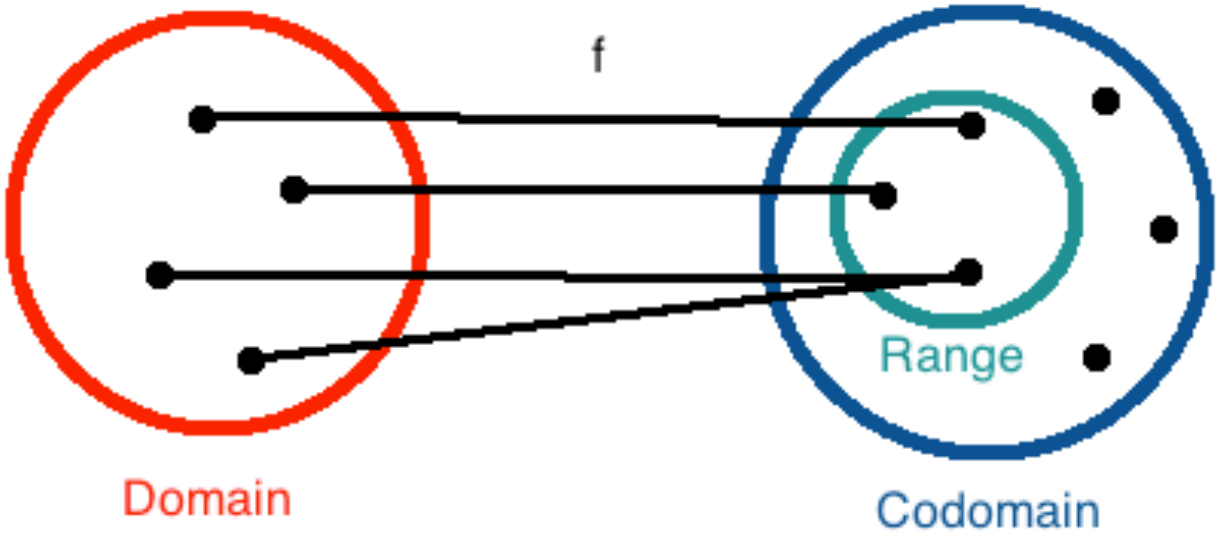
an aexpression (aexp) is
  either *simple*, and has [*CAth*231"
    a *body* which is an identifier
  or a *combination*, in which case it has [*sin*(*a*+2*b*)
    a *rator*, which is an aexp, or
    and a *rand*, which is an aexp, *a* + 2*b*
  or *conditional*, in which case it is
    either *two-armed*, and has [*p*→*a*+2*b*; 2*a*−*b*
      a *condition*, which is an aexp,
      and a *leftarm*, which is an aexp,
      and a *rightarm*, which is an aexp,
    or *one-armed*, and has [*q*→2*a*−*b*
      a *condition*, which is an aexp,
      and an *arm*, which is an aexp,
  or a *listing*, and has [*a*+*b*, *c*+*d*, *e*+*f*
    a *body* which is an aexp-list,
  or *beet*, and has [*x*(*x*+1) **where** *x* = a + 2*b*
    a *mainclause*, which is an aexp, or
    and a *support* **let** *x* = *a* + 2*b*; *x*(*x*+1)
      which is an adef,

an adefinition (adef) is
  either *standard*, and has [$x=a+2b$
    a *definee* (*nee*), which is an abv,
    and a *definiens* (*niens*), which is an aexp,
  or *functionform*, and has [$f(x)=x(x+1)$
    a *lefthandside* (*lhs*),
      which is an abv-list of length ≥ 2,
    and a *righthandside* (*rhs*), which is an aexp
  or *programpoint*, and has [**pp** $f(x)=x(x+1)$
    a *body* which is an adef,
  or *circular*, and has [**rec** $f(n)=(n=0)\rightarrow 1;\quad nf(n-1)$
    a *body* which is an adef,
  or *simultaneous*, and has [$x=a+2b$ **and** $y=2a-b$
    a *body*, which is an adef-list,
  or *beet*, and has [$f(y)=x(x+y)$
    a *mainclause*, **where** $x=a+2b$
    which is an adef,
    and a *support*, which is an adef.
where an abv is
  either *simple*, and has
    a *body*, which is an identifier,
  or **else**, is an abv-list. [$x,\ (y,\ z),\ w$

Take-aways

- Lambda calculus was the first programming language.
- Programs and data in lambda calculus are treated uniformly, in that both are ‘first-class’.
- Landin gave lambda calculus a form (ISWIM) that resembled existing math notation and existing programming languages, but was small.
- Landin described a ‘runtime’ that could be used to execute his language that was abstracted away from the hardware machine.

Why do we need IF?



Function

$$f(x) = |x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$$

Piecewise function

$$f(x) = \frac{x+1}{x^2 - 3x + 2}$$

Partial function

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their

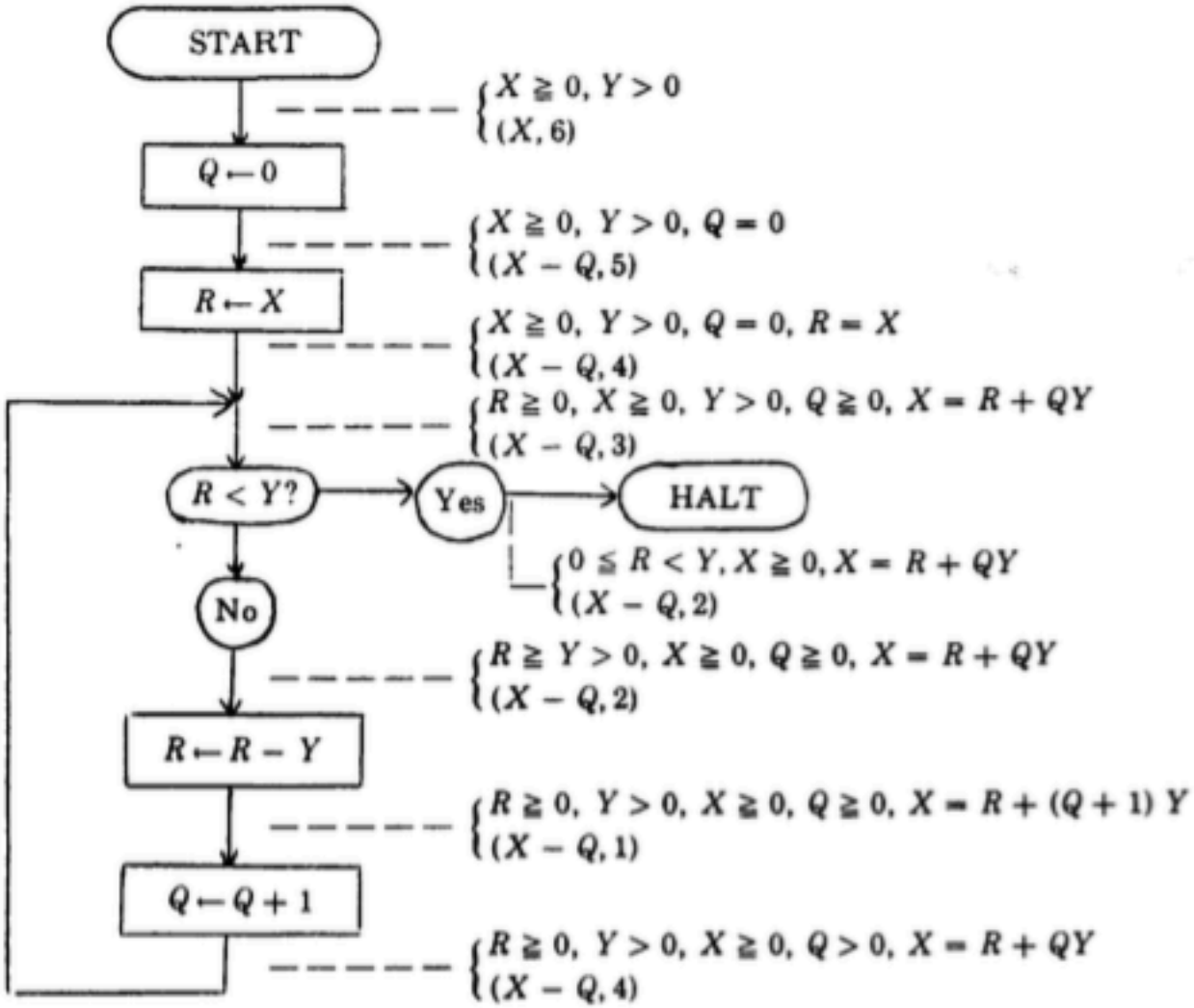


FIGURE 5. Algorithm to compute quotient Q and remainder R of $X \div Y$, for integers $X \geq 0$, $Y > 0$

"Are these things a proof aid or are they implemented when you run the program? I take it they were the former and not the latter”

- Peter Landin, recollecting attending a Floyd lecture

```
public void setRefreshRate(int rate) {
  // Enforce specified precondition in public method
  if (rate <= 0 || rate > MAX_REFRESH_RATE)
    throw new IllegalArgumentException("Illegal rate: " + rate);
    setRefreshInterval(1000/rate);
  }
```

```
private void setRefreshInterval(int interval) {
 // Confirm adherence to precondition in nonpublic method
 assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;

 ... // Set the refresh interval
}
```

```
public BigInteger modInverse(BigInteger m) {
  if (m.signum <= 0)
    throw new ArithmeticException("Modulus not positive: " + m);
  ... // Do the computation
  assert this.multiply(result).mod(m).equals(ONE) : this;
  return result;
}
```

http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

"What would we think of a sailing enthusiast who wears his life jacket when training on dry land, but takes it off as soon as he goes to sea?”

- Tony Hoare

Hints on Programming Language Design

```ruby
def sign_in_and_setup
  signed_in_user

  unless current_user.nil?
    # Initialize objects

      •

      •

      •

    @enrollment = get_enrollment(current_user)
    @enrollment = current_enrollment unless current_enrollment.nil? || @enrollment
  end
end
end
```

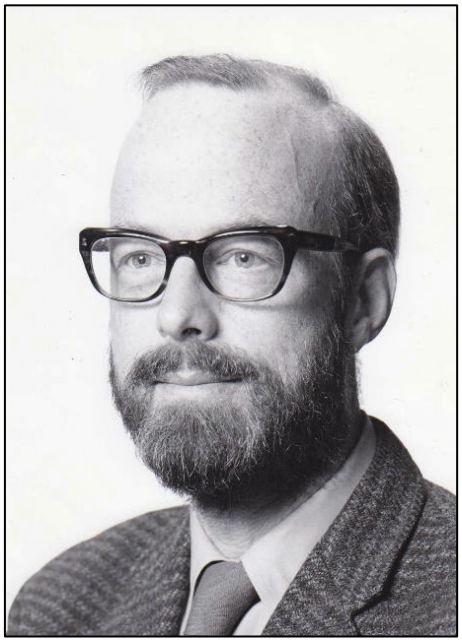
“I call it my billion dollar mistake”

- Tony Hoare
QCon 2009

RECORD HANDLING

C. A. R. Hoare

A series of lectures to be delivered at
The NATO Summer School,
Villard-de-Lans
September 12 - 16, 1966.



2.2 Partial Functional Relationships.

Functional relationships may be classified as either partial or total. A total functional relationship is one which satisfies the condition that for every x in the class which defines its domain, there is exactly one y in its range to which x bears the relationship. For a partial functional relationship, there may be an x for which there is no y appropriately related to it. All the relationships introduced above for persons are in fact partial, since there will be persons who have no offspring, and persons who have no elder brother or sister; and in any finite collection of persons, there must be at least one who has no father. In order to meet this problem, a special null value is provided for reference variables and fields. If a field of a record is given this value, it usually indicates that the functional relationship represented by that field is not defined (or not yet defined) for that record; and that it is therefore a partial rather than total functional relationship. The null reference value is denoted by the basic symbol null.

II. Notes on Data Structuring *

C. A. R. HOARE

1. INTRODUCTION

In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities, and to ignore for the time being the differences. As soon as we have discovered which similarities are relevant to the prediction and control of future events, we will tend to regard the similarities as fundamental and the differences as trivial. We may then be said to have developed an abstract concept to cover the set of objects or situations in question. At this stage, we will usually introduce a word or picture to symbolise the abstract concept; and any particular spoken or written occurrence of the word or picture may be used to *represent* a particular or general instance of the corresponding situation.

The primary use for representations is to convey information about important aspects of the real world to others, and to record this information in written form, partly as an aid to memory and partly to pass it on to future generations. However, in primitive societies the representations were sometimes believed to be useful in their own right, because it was supposed that manipulation of representations might in itself cause corresponding changes in the real world; and thus we hear of such practices as sticking pins into wax models of enemies in order to cause pain to the corresponding part of the real person. This type of activity is characteristic of magic and witchcraft. The modern scientist on the other hand, believes that the manipu-

THE FORMULAE-AS-TYPES NOTION OF CONSTRUCTION

W. A. Howard

Department of Mathematics, University of Illinois at Chicago Circle, Chicago, Illinois 60680, U.S.A.

Dedicated to H. B. Curry on the occasion of his 80th birthday.

The following consists of notes which were privately circulated in 1969. Since they have been referred to a few times in the literature, it seems worth while to publish them. They have been rearranged for easier reading, and some inessential corrections have been made.

The ultimate goal was to develop a notion of construction suitable for the interpretation of intuitionistic mathematics. The notion of construction developed in the notes is certainly too crude for that, so the use of the word *construction* is not very appropriate. However, the terminology has been kept in order to preserve the original title and also to preserve the character of the notes. The title has a second defect; namely, a type should be regarded as a abstract object whereas a *for-*

$$(\to\text{-intro})\quad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma.\, e) : \sigma \to \tau}$$

$$(\to\text{-elim})\quad \frac{\Gamma \vdash \phi_1 \Rightarrow \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2} \qquad \frac{\Gamma \vdash e_0 : \sigma \to \tau \quad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash (e_0\ e_1) : \tau}$$

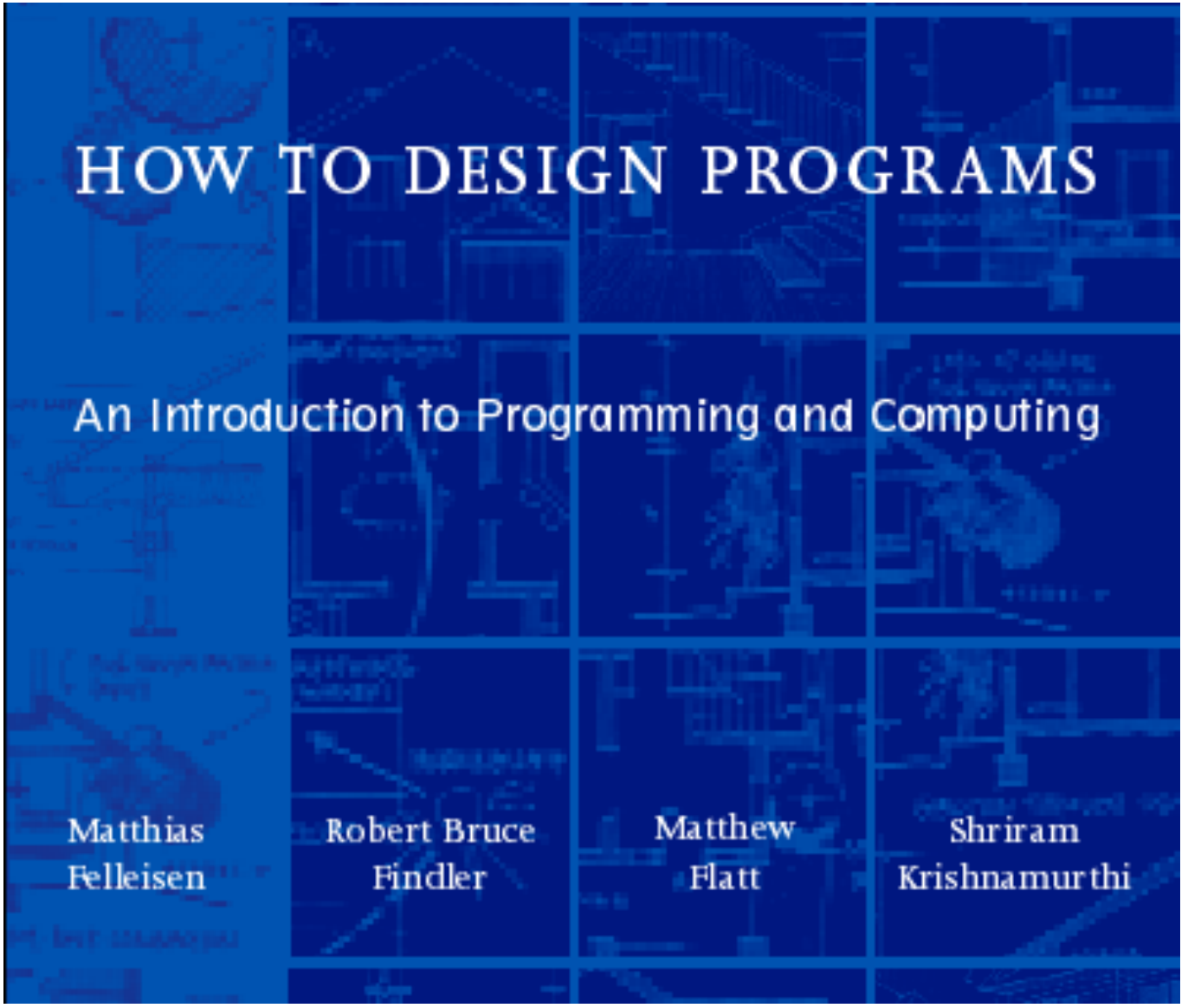
$$(\wedge\text{-intro})\quad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma * \tau}$$

$$(\wedge\text{-elim})\quad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \quad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \qquad \frac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#1\, e : \sigma} \quad \frac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#2\, e : \tau}$$

$$(\vee\text{-intro})\quad \frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \quad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathbf{inl}_{\sigma+\tau} : e\sigma + \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{inr}_{\sigma+\tau} : e\sigma + \tau}$$

$$(\vee\text{-elim})\quad \frac{\Gamma \vdash \phi \vee \psi \quad \Gamma \vdash \phi \to \chi \quad \Gamma \vdash \psi \to \chi}{\Gamma \vdash \chi} \qquad \frac{\Gamma \vdash e : \sigma + \tau \quad \Gamma \vdash e_1 : \sigma \to \rho \quad \Gamma \vdash e_2 : \tau \to \rho}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ e_1 \mid e_2 : \rho}$$

Demo in Pyret


HOW TO DESIGN PROGRAMS
An Introduction to Programming and Computing
Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi

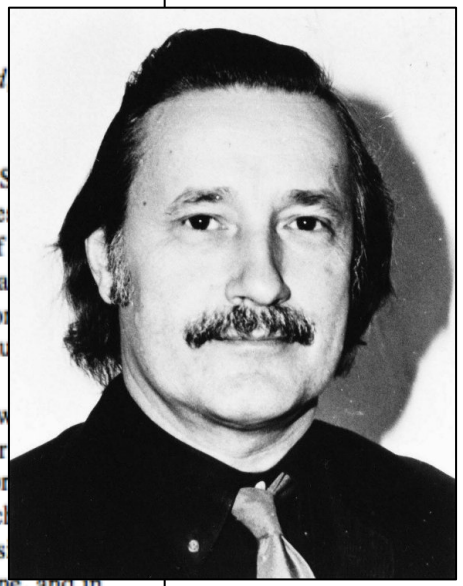
Recipe

1. Understand how to represent information as a data definition.
2. Write down function header and purpose.
3. Write example calls (test cases).
4. Put template for data definition in body.
5. Fill the template with code.
6. Run the tests and fix.

Demo in Pyret

Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford



Abstract. This paper forms the substance of a course of lectures given at the International Summer S
Computer Programming at Copenhagen in August, 1967. The lectures were originally given from note
paper was written after the course was finished. In spite of this, and only partly because of the shortage of
paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of a
never quite clear what sort of audience there will be for such lectures—and an associated switching fro
to informal modes of presentation which may well be less acceptable in print than it is natural in the lectu
For these (and other) faults, I apologise to the reader.

There are numerous references throughout the course to CPL [1–3]. This is a programming language w
been under development since 1962 at Cambridge and London and Oxford. It has served as a vehicle for
into both programming languages and the design of compilers. Partial implementations exist at Cambr
London. The language is still evolving so that there is no definitive manual available yet. We hope to reach
resting point in its evolution quite soon and to produce a compiler and reference manuals for this vers
compiler will probably be written in such a way that it is relatively easy to transfer it to another machine, and in
the first instance we hope to establish it on three or four machines more or less at the same time.

The lack of a precise f
concerned with the idea
language.

Keywords: programm
meter passing, variable
mechanisms, type comp

Parametric polymorphism is more regular and may be illustrated by an example. Suppose `f` is a function whose argument is of type α and whose results is of β (so that the type of `f` might be written $\alpha \Rightarrow \beta$), and that `L` is a list whose elements are all of type α (so that the type of `L` is α **`list`**). We can imagine a function, say `Map`, which applies `f` in turn to each member of `L` and makes a list of the results. Thus `Map[f,L]` will produce a β **`list`**. We would like `Map` to work on all types of list provided f was a suitable function, so that `Map` would have to be polymorphic. However its polymorphism is of a particularly simple

1. Preliminaries

1.1. Introduction

Any discussion on the foundations of computing runs into severe problems right at the

Demo in Pyret

```
if (do-you-know-what-time-it-is()) then
  tell-me-the-time()
else
 ….
end
```

"The branches of the conditional are both blind to the outcome and to the meaning of the boolean value computed by the test"

```
case (whattimeisit()) of
SOME t => ...
NONE => ...
```

Boolean Blindness

Demo in Pyret

Not Covered

Abstract data types –
Selection is hidden in the implementation. CLU advanced this approach to building larger systems. SML modules extended it. Interfaces retain some of the flavor of abstract types.

Dependent types –
Selection semantics moved from term logic to
type system. Can be statically analyzed by type checker.

Notes

- Conditional logic in programming can be viewed through levels of abstraction.
- We will try to split into computational abstraction and informational abstraction
- The (over) reliance on the IF statement is part historical accident, part need for speed requiring low-level constructs, and part stalled evolution of programming languages.
- Introduction of conditional *expression* provided more mathematical programming model based on general recursive functions. This turned out to be important because static analysis of imperative programming using assertions and Hoare Logic didn't work out, despite decades of trying.
- LISP showed that programming didn't need to mimic the hardware execution.
- Teaching students to be human compilers and CPUs is unnecessary.

Notes

- First-class functions, particularly higher-order functions, enabled by lambda calculus model of computation allow greater generality and for selection in some cases to move to the function caller.
- ISWIM as a small lambda calculus core, with use of libraries to extend the language, changes what it means to learn a programming language.
- ISWIM's SECD virtual machine abstracted away the hardware altogether, allowing the programming model to not have to be polluted by the hardware model.
- I want to emphasize the need for a "runtime" to separate a hostile OS from the programming.

Notes

- Conditionals represent *piecewise functions*, in which the mapping from domain to codomain varies according to properties of elements in the domain.
- Programming is often problematic because the functions (ie, the mapping of one set of elements to another set of elements) are *partial functions*, not defined for all elements in the domain.
- The effort to trace the conditions under which a program is defined began with Floyd in 1968 and developed into attempt to specify the behavior through series of assertions. But they were not machine checked.
- Runtime assertions are still featured in some languages, but clutter the code and slow down its execution, so are typically turned off. And they just blow up the program anyway.
- IF is used on ad hoc basis to emulate assertions. A common use case is checking for NULL. Another is unit tests, which can only handle the topmost *preconditions*. They don't address the semantics of the program.

Notes

- NULL was introduced by Hoare in influential 1966 paper. Records became incorporated into Simula 67, which was the foundation for OOP.
- Opinion: the branch of OOP which relied on late binding, namely Smalltalk, isn't suitable for large projects. Won't discuss this form of selection.
- Hoare summarized data structuring in 1970, including *discriminated union* (a.k.a tagged union, variant type, sum type), although it was in Algol 68 already.
- The discriminated union would be in CLU, Ada, Modula 2. But is not found in more recent OO languages. Presumably due to the mistaken assumption that subtyping was sufficient to represent it.
- Won't cover Expression Problem, although it is interesting for notion of extension, including in "late binding languages". I believe the notion of extension is suspect in practice.

Notes

- Bill Howard (re)discovered a curious relationship between logic and typing rules in 1969 (but didn't publish until 1980).
- The logic operators for implication, conjunction, and disjunction correspond exactly to the typing rules for functions, records, and discriminated unions.
- They all come in pairs (so-called *introduction* of the form and *elimination* of it).
- Pattern matching of discriminated union types is the elimination form.
- Why is this not just an academic curiosity?
- *How To Design Programs* (Felleisen et al) is a systematic process of building programs by deliberate representation of information and using focused functions to destructure and transform it. The introduction/elimination approach is even more apparent in the Pyret language.
- HTDP is at a high level of abstraction that is closer to the problem domain of interest, which helps make it appealing to beginners.
- HTDP produces *good* programs.

Notes

- Strachey introduced the notion of *parametric polymorphism* informally in 1967.
- When function parameters are polymorphic in their type, then the properties of the computation can be understood independent of concrete representations. This is a huge deal because now we can find and represent patterns (which are part of the computation itself, not the process).
- In solving for a specific problem domain, we can now find patterns and properties, instead of relying on the domain jargon for conceptual models.
- The need for selection under parametric polymorphism is essentially removed.

Notes

- With parametric polymorphism and elimination form of discriminated unions, we can now understand *boolean blindness*.
- Relying on predicate tests results in loss of information, which has to be recovered in some other way. The branches of a conditional are *blind* to information that is being dispatched on. There is little to prevent the branch from doing the wrong thing.
- Encoding the information in the type (parametric discriminated union) retains it through the computation.