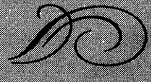
Selected Papers on Computer Languages



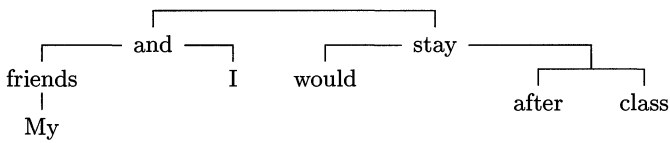
DONALD E. KNUTH

Contents

Preface

This book is devoted to the topic that got me hooked on computers in the first place: the languages by which people are able to communicate with machines.

Rules of grammar have always fascinated me. In retrospect I think that the most exciting days of my elementary school education came during the seventh grade, when I was taught how to make diagrams of English-language sentences in order to see how the various parts of



speech fit together. My friends and I would stay after class, trying to put more and more sentences into diagrammatic form — and getting totally confused by poetry, which seemed to break most of the rules.

During the summer of 1957, after I had just written my first computer programs for numerical problems like factoring numbers and for fun problems like playing tic-tac-toe, an amazing program called IT — the "Internal Translator" — arrived at the Case Tech Computing Center, where I had been working as a college freshman. IT would look at a card that contained an algebraic equation, then the machine would flash its lights for a few seconds and go punch-punch-punch; out would come a deck of cards containing a machine-language program to compute the value of the equation. Mystified by the fact that a mere computer could understand algebra well enough to create its own programs, I got hold of the source code for IT and spent two weeks poring over the listing. Lo, the secrets were revealed, and several friends helped me in 1958 to extend IT even further.

Of course in those days we had only a vague notion of what we were doing; our work was almost totally disorganized, with very few principles to guide us. But researchers in linguistics were beginning to formulate rules of grammar that were considerably more mathematical than before. And people began to realize that such methods are highly relevant to the artificial languages that were becoming popular for computer programming, even though natural languages like English remained intractable.

I found the mathematical approach to grammar immediately appealing — so much so, in fact, that I must admit to taking a copy of Noam Chomsky's *Syntactic Structures* along with me on my honeymoon in 1961. During odd moments, while crossing the Atlantic in an ocean liner and while camping in Europe, I read that book rather thoroughly and tried to answer some basic theoretical questions. Here was a marvelous thing: a mathematical theory of language in which I could use a computer programmer's intuition! The mathematical, linguistic, and algorithmic parts of my life had previously been totally separate. During the ensuing years those three aspects became steadily more intertwined; and by the end of the 1960s I found myself a Professor of Computer Science at Stanford University, primarily because of work that I had done with respect to languages for computer programming.

The present book begins with an account of what was happening all over the world before I myself had ever heard about programming languages. Chapter 1 tells the story of more than twenty pioneering efforts to design and implement notations and languages that could help programmers cope with the first computers. These intriguing and instructive developments represent the first creative steps in the evolution of "software," forming a parallel thread to the better-known history of hardware development.

ALGOL 60, the international algebraic language that profoundly influenced all subsequent work, is the concern of Chapters 2–8. Brief introductory remarks in Chapters 2 and 3 about ALGOL's basic concepts, especially about the important way in which the grammatical rules of ALGOL were presented, are followed by Chapter 4, a brash and somewhat hastily written article in which I recorded the initial reactions that my colleagues and I were having to ALGOL's more controversial innovations. Chapter 5 describes SMALGOL-61, a subset of ALGOL 60 intended to respect the limitations of modest-size machines. One of the most subtle aspects of the full language, known as "call-by-name recursion" and "up-level addressing," is considered briefly in Chapter 6. Then Chapter 7 presents a (short-lived yet interesting) proposal for expressing input and output operations at a high level, without changing

the underlying language, although ALGOL 60 itself did not provide any such facilities. The preparation of that proposal gave me first-hand experience with the process of design-by-committee; it also led to a simple concept called a *list procedure* that deserves to be more widely known. Chapter 8 surveys the "remaining trouble spots" of ALGOL, which turn out to be remarkably few.

My own first attempt at designing a computer language — SOL, the Simulation Oriented Language — is the subject of Chapters 9 and 10. Shortly after doing that work I drafted some notes about the general principles of language design, intending to include them as part of a survey paper that I had been asked to write. Those notes, never before published, still seem surprisingly relevant, so I have included them as Chapter 11.

Chapter 12 points out that programming languages can be used for theoretical purposes to study abstract computers, just as they are used on real machines. Chapter 13 is a purely theoretical paper for which I have fond memories, because it solves a problem that had stumped me for several weeks during the 1960s. Finally I had decided that it was too tough for me, and resumed working on other things ... but when I awoke the next day, aha! The solution had somehow magically popped into my head. Although the problem itself is mathematically elegant, it is rather technical and has no known practical application; yet I found the work extremely instructive, and the things I learned while seeking a solution proved to be valuable in many subsequent investigations. Thus I came to appreciate the benefits that theory brings to practice.

The focus shifts in Chapter 14 to questions of *parsing*, namely to the study of methods by which a computer can efficiently recognize the grammatical structure of sentences that it reads. Volume 5 of my books on *The Art of Computer Programming* (*TAOCP*) is scheduled to be largely about parsing, and the material on "top-down parsing" in Chapter 14 of the present book is based largely on the draft manuscript of *TAOCP* as it existed in 1964. (I didn't publish any of those elementary ideas at the time, because I was optimistic that *TAOCP* would soon be in print.) Shortly after completing that draft, I realized that many of the ideas I had been reading about and discussing could be unified and extended, and that the resulting "bottom-up" approach would be quite powerful. This new approach seemed too advanced for my book, so I took time out to submit it for publication in 1965. The result, called LR(k) parsing because it involves "translation from left to right by looking k characters ahead," appears in Chapter 15. (Many people went on to refine and simplify the associated theory, which

then became integrated into software practice; so I shall certainly have to discuss it in depth when I finally do finish the manuscript of Volume 5.)

Chapter 16 comes from the draft of Volume 6. It contains basic ideas about the theory of languages that I developed in 1964 but didn't publish until 1992, namely that the theory becomes even better if we think of languages as *multisets* of sentences, not just as ordinary sets.

Once a language can be parsed, we can attach *meaning* to its phrases, thereby complementing syntax with semantics. Chapters 17–19 deal with my favorite approach to this important subject, which is now popularly known as the study of *attribute grammars.*

The next few chapters contain background material that is largely of historical interest. Chapter 20 portrays the problems of parsing and translation as they were viewed in 1962; it explains why I was so delighted in later years to help develop a solid theoretical basis by which compiler techniques could be systematically understood and improved. Chapter 21 — my very first paper about computing, written when I was an undergraduate — provides an excellent contrast to the more "mature" efforts that followed, and contains an incredibly complicated flowchart. Flowcharts are the subject of Chapter 22; the experiments discussed in that chapter led to the method of exposition that I subsequently decided to adopt in *TAOCP.* Some early thoughts about "structured programming," as an alternative to flowcharts, appear in Chapter 23.

The emphasis shifts again in Chapter 24, towards the generation of efficient low-level machine code from high-level source language input. This chapter launched the empirical study of "typical" user programs, and recommended *profiling* tools that help speed programs up.

Finally, Chapter 25 — my most recent paper — considers another sort of optimization that curiously has not been addressed before, based on the transformation of *coroutines.* This chapter demonstrates that computer languages still contain fundamental aspects that are not well understood; I suspect that a thorough study of coroutine transformation will lead to further discoveries of important ideas.

I'm dedicating this book to my mother, Louise Knuth, whose deep love of language proved to be contagious. She encouraged me to read at a very early age, and continued to inspire me with her boundless energy. On the occasion of her 80th birthday in October, 1992, I began my after-dinner remarks as follows:

> Four score and zero years ago, my grandparents brought forth on this continent a new baby, conceived in Ohio, and dedicated

to prepositions, adjectives, nouns, adverbs, and all other kinds of words.

I'm also dedicating this book to my dear friend Ole-Johan Dahl, whose deep insights into computational processes led to the important notion of object-oriented languages. When I learned in 1966 of the SIMULA language that he developed together with Kristen Nygaard, it was love at first sight, and I immediately abandoned any further work with SOL. Ole-Johan has written some of the most elegant computer programs that I have ever seen. He gave me the opportunity to work with him for a year in Norway, where I learned many things that inspired my subsequent work. Fate decreed, alas, that he and my mother would both die during the same week, last summer; but the memory of their productive lives has sustained me as I prepared this book for publication.

I'm grateful too for having been able to work jointly with coauthors Richard Bigelow, Bob Floyd, John McNeley, Jack Merner, Frank Ruskey, and Luis Trabb Pardo. My research was generously supported by Burroughs Corporation, by IBM Corporation, and by various agencies of the U.S. government.

The present book has been made possible by the skills of Max Etchemendy, Tony Gee, Lauri Kanerva, Kimberly Lewis Brown, and Christine Sosa, who helped to convert hundreds of pages published long, long ago into a modern electronic format. Dan Eilers, Mark Ward, and Udo Wermuth gave invaluable help with the nitty-gritty details of proofreading. Papers on computer languages surely rank among the most difficult challenges that printers have ever had to face, from a typographic standpoint, so it gives me great pleasure to see this material finally typeset in the way I originally intended. (Perhaps even better.)

The process of compiling this book has given me the incentive to improve some of the original wording. Indeed, I first wrote nearly all of these chapters long before the copy editors of *TAOCP* taught me how to write better sentences. I cannot be happy now with stylistic errors that I would no longer tolerate in the writing of my students, so I have tried to remove them, together with all of the technical errors that have come to my attention. On the other hand I have attempted to retain the essence of the original exposition, and to avoid injecting any hindsight into the texts, so as not to imply falsely that I was clairvoyant or that I had anticipated certain ideas or trends sooner than other people did.

The only place where I found it necessary to change my original notation and terminology significantly was in Chapter 15, which was written when the theory of context-free languages was still in a state of

flux: I've changed the term "intermediate symbol" to the now-standard term "nonterminal symbol" throughout that article. Moreover, Chapter 15 was written during a period when I still had not decided how to draw the tree diagrams in *TAOCP*; should they have the root at the bottom, as in nature, or at the top, as many of my friends were drawing them? [See Figure 2–17 in Volume 1.] I experimented for awhile with a return to nature, but I soon realized that common terms like "top-down parsing" gave me no choice. Thus, the nature-oriented tree diagrams in my original paper on LR(k) grammars are needlessly confusing today.

Nearly all of the illustrations in this book — more than 75 of them — have, incidentally, been redrawn using John Hobby's METAPOST, a computer language that continues to give me great joy.

Most of the chapters now also have an Addendum, in which I try to bring everything up to date by discussing the most important sequels to the original work, as far as I know them. And I have tried to put all of the bibliographies into a consistent format, so that they will be as useful as possible to everyone who pursues the subject.

This book is number five in a series that Stanford's Center for the Study of Language and Information (CSLI) plans to publish containing archival forms of the papers I have written. The first volume, *Literate Programming*, appeared in 1992; the second, *Selected Papers on Computer Science*, appeared in 1996; the third, *Digital Typography*, in 1999; and the fourth, *Selected Papers on Analysis of Algorithms*, in 2000. Three additional volumes are in preparation containing selected papers on Discrete Mathematics, Design of Algorithms, Fun and Games.

Donald E. Knuth
Stanford, California
March 2003

Acknowledgments

"The early development of programming languages" originally appeared in *Encyclopedia of Computer Science and Technology* **7** (New York: Marcel Dekker, Inc., 1977), pp. 419–493. Copyright ©1977 by Marcel Dekker, Inc. Reprinted by permission.

"Backus Normal Form versus Backus Naur Form" originally appeared in *Communications of the ACM* **7**, (December 1964), pp. 735–736. (New York: Association for Computing Machinery, Inc.) Copyright presently held by the author.

"Teaching ALGOL 60" originally appeared in *Algol Bulletin* **19** (Amsterdam: Mathematisch Centrum, January 1965), pp. 4–6. Copyright presently held by the author.

"ALGOL 60 Confidential" originally appeared in *Communications of the ACM* **4** (June 1961), pp. 268–272. (New York: Association for Computing Machinery, Inc.) Copyright presently held by the author.

"SMALGOL-61" originally appeared in *Communications of the ACM* **4** (November 1961), pp. 499–502. (New York: Association for Computing Machinery, Inc.) Copyright presently held by the author.

"Man or boy?" originally appeared in *Algol Bulletin* **17** (Amsterdam: Mathematisch Centrum, July 1964), p. 7; **19** (January 1965), pp. 8–9. Copyright presently held by the author.

"A proposal for input-output conventions in ALGOL 60" originally appeared in *Communications of the ACM* **7** (May 1964), pp. 273–283. (New York: Association for Computing Machinery, Inc.) Copyright presently held by the author.

"The remaining trouble spots in ALGOL 60" originally appeared in *Communications of the ACM* **10** (October 1967), pp. 611–618. (New York: Association for Computing Machinery, Inc.) Copyright presently held by the author.

"SOL — A symbolic language for general purpose systems simulation" originally appeared in *IEEE Transactions on Electronic Computers* **EC-13** (1964), pp. 401–408. Copyright ©1964 by The Institute of Electrical and Electronics Engineers, Inc. Reprinted by permission.

"A formal definition of SOL" originally appeared in *IEEE Transactions on Electronic Computers* **EC–13** (1964), pp. 409–414. Copyright ©1964 by The Institute of Electrical and Electronics Engineers, Inc. Reprinted by permission.

"Programming languages for automata" originally appeared in *Journal of the ACM* **14** (October 1967), pp. 615–635. (New York: Association for Computing Machinery, Inc.) Copyright presently held by the author.

"A characterization of parenthesis languages" originally appeared in *Information and Control* **11** (1967), pp. 269–289. Copyright ©1967 by Academic Press. All rights reserved. Reprinted by permission.

Lou Knuth with half of her clan, January 2002
(photo by Diana Baxter)

Chapter 1

The Early Development of Programming Languages

[Written with Luis Trabb Pardo. Originally published in Encyclopedia of Computer Science and Technology, edited by Jack Belzer, Albert G. Holzman, and Allen Kent, Volume 7 (New York: Marcel Dekker, 1977), 419–493.]

This paper surveys the evolution of "high-level" programming languages during the first decade of computer programming activity. We discuss the contributions of Zuse in 1945 (the "Plankalkül"), Goldstine and von Neumann in 1946 ("Flow Diagrams"), Curry in 1948 ("Composition"), Mauchly et al. in 1949 ("Short Code"), Burks in 1950 ("Intermediate PL"), Rutishauser in 1951 ("Klammerausdrücke"), Böhm in 1951 ("Formules"), Glennie in 1952 ("AUTOCODE"), Hopper et al. in 1953 ("A-2"), Laning and Zierler in 1953 ("Algebraic Interpreter"), Backus et al. in 1954–1957 ("FORTRAN"), Brooker in 1954 ("Mark I AUTOCODE"), Kamynin and Liûbimskiĭ in 1954 ("ПП-2"), Ershov in 1955 ("ПП"), Grems and Porter in 1955 ("BACAIC"), Elsworth et al. in 1955 ("Kompiler 2"), Blum in 1956 ("ADES"), Perlis et al. in 1956 ("IT"), Katz et al. in 1956–1958 ("MATH-MATIC"), Bauer and Samelson in 1956–1958 (U.S. Patent 3,047,228).

The principal features of each contribution are illustrated and discussed. For purposes of comparison, a particular fixed algorithm has been encoded as far as possible in each of the languages.

This summary of early work is based primarily on unpublished source materials, and the authors hope that they have been able to compile a fairly complete picture of the pioneering developments in this area.

Introduction

It is interesting and instructive to study the history of a subject not only because it helps us to understand how the important ideas were born—

and to see how the "human element" entered into each development — but also because it helps us to appreciate the amount of progress that has been made. This is especially striking in the case of programming languages, a subject that has long been undervalued by computer scientists. After learning a high-level language, a person often tends to think mostly of improvements that he or she would like to see, since all languages can be improved, and it is very easy to underestimate the difficulty of creating that language in the first place. In order to perceive the real depth of this subject, we need to realize how long it took to develop the important concepts that we now regard as self-evident. The ideas were by no means obvious a priori, and many years of work by brilliant and dedicated people were necessary before our current state of knowledge was reached.

The goal of this paper is to give an adequate account of the early history of high-level programming languages, covering roughly the first decade of their development. Our story will take us up to 1957, when the practical importance of algebraic compilers was first being demonstrated, and when computers were just beginning to be available in large numbers. We will see how people's fundamental conceptions of algorithms and of the programming process evolved during the years — not always in a forward direction — culminating in languages such as FORTRAN I. The best languages we shall encounter are, of course, very primitive by today's standards, but they were good enough to touch off an explosive growth in language development; the ensuing decade of intense activity has been detailed in Jean Sammet's 785-page book [SA 69]. We shall be concerned with the more relaxed atmosphere of the "pre-Babel" days, when people who worked with computers foresaw the need for important aids to programming that did not yet exist. In many cases these developments were so far ahead of their time that they remained unpublished, and they are still largely unknown today.

Altogether we shall be considering about 20 different languages; therefore we will have neither the space nor the time to characterize any one of them completely. Besides, it would be rather boring to recite so many technical rules. The best way to grasp the spirit of a programming language is to read example programs, so we shall adopt the following strategy: A certain fixed algorithm — which we shall call the "TPK algorithm" for want of a better name[1] — will be expressed as a program in each language we discuss. Informal explanations of this

[1] Consider "Grimm's law" in comparative linguistics, and/or the word "typical," and/or the names of the authors of this article.

program should then suffice to capture the essence of the corresponding language, although the TPK algorithm will not, of course, exhaust that language's capabilities. Once we have understood a program for TPK, we'll be able to discuss the most important language features that it does not reveal.

Note that the same algorithm will be expressed in each language, in order to provide a simple means of comparison. A serious attempt has been made to write each program in the style originally used by the authors of the corresponding languages. If comments appear next to the program text, they attempt to match the terminology used at that time by the original authors. Our treatment will therefore be something like a recital of "Chopsticks" as it would have been played by Bach, Beethoven, Brahms, and Brubeck. The resulting programs are not truly authentic excerpts from the historic record, but they will serve as fairly close replicas; the interested reader can pursue each language further by consulting the bibliographic references to be given.

The exemplary TPK algorithm that we shall be using so frequently can be written as follows in a dialect of ALGOL 60:

```
01    TPK: begin integer i; real y; real array a[0:10];
02         real procedure f(t); real t; value t;
03           f := sqrt(abs(t)) + 5 × t ↑ 3;
04         for i := 0 step 1 until 10 do read(a[i]);
05         for i := 10 step −1 until 0 do
06           begin y := f(a[i]);
07             if y > 400 then write(i, 'TOO LARGE')
08                        else write(i, y);
09           end
10       end TPK.
```

[Actually ALGOL 60 is not one of the languages we shall be discussing, since it was a later development, but the reader ought to know enough about it to understand TPK. If not, here is a brief run-down on what the program means: Line *01* says that i is an integer-valued variable, whereas y takes on floating point approximations to real values; and a_0, a_1, ..., a_{10} also are real-valued. Lines *02* and *03* define the function $f(t) = \sqrt{|t|}+5t^3$ for use in the algorithm proper, which starts on line *04*. Line *04* reads in eleven input values a_0, a_1, ..., a_{10}, in this order; then line *05* says to do lines *06*, *07*, *08*, *09* (delimited by **begin** and **end**) for $i = 10, 9, \ldots, 0$, in *that* order. The latter lines cause y to be set to $f(a_i)$, and then one of two messages is written out. The message is

either the current value of i followed by the words 'TOO LARGE', or the current values of i and y, according as $y > 400$ or not.]

Of course this algorithm is quite useless; but for our purposes it will be helpful to imagine ourselves vitally interested in the process. Let us pretend that the function $f(t) = \sqrt{|t|} + 5t^3$ has a tremendous practical significance, and that it is extremely important to print out the function values $f(a_i)$ in the opposite order from which the a_i are received. This will put us in the right frame of mind to be reading the programs below. (If a truly useful algorithm were being considered here, it would need to be much longer in order to illustrate as many different features of the programming languages.)

Many of the programs we shall discuss will have italicized line numbers in the left-hand margin, as in the ALGOL code above. Such numbers are not really part of the programs; they appear only so that the accompanying text can refer easily to any particular line.

It turns out that most of the early high-level languages were not able to handle the TPK algorithm exactly as presented above; therefore we must make some modifications. In the first place, when a language deals only with integer variables, we shall assume that all inputs and outputs are integer valued, and that '$sqrt(x)$' denotes the largest integer not exceeding $\sqrt{x}$. Second, if the language does not provide for alphabetic output, the string 'TOO LARGE' will be replaced by the number 999. Third, some languages do not provide for input and output at all; in such a case, we shall assume that the input values a_0, a_1, ..., a_{10} have somehow been supplied by an external process, and that our job is to compute 22 output values b_0, b_1, ..., b_{21}. Here b_0, b_2, ..., b_{20} will be the respective "i values" 10, 9, ..., 0, and the alternate positions b_1, b_3, ..., b_{21} will contain the corresponding $f(a_i)$ values and/or 999 codes. Finally, if a language does not allow programmers to define their own functions, the statement '$y := f(a[i])$' will essentially be replaced by its expanded form '$y := sqrt(abs(a[i])) + 5 \times a[i] \uparrow 3$'.

Prior Developments

Before getting into real programming languages, let us try to set the scene by reviewing the background very quickly. How were algorithms described prior to 1945?

The earliest known written algorithms come from ancient Mesopotamia, about 2000 B.C. In this case the written descriptions contained only sequences of calculations on particular sets of data, not an abstract statement of a particular procedure. It is clear that strict procedures

were being followed (since, for example, multiplications by 1 were explicitly performed), but they never seem to have been written down. Iterations like '**for** $i := 0$ **step** 1 **until** 10' were rare, but when present they would consist of a fully expanded sequence of calculations. (See [KN 72] for a survey of Babylonian algorithms.)

By the time of Greek civilization, several nontrivial abstract algorithms had been studied rather thoroughly. For example, see [KN 69, §4.5.2] for a paraphrase of Euclid's presentation of "Euclid's algorithm." The description of algorithms was always informal, however, rendered in natural language.

Mathematicians never did invent a good notation for dynamic processes during the ensuing centuries, although of course notations for (static) functional relations became highly developed. When a procedure involved nontrivial sequences of decisions, the available methods for precise description remained informal and rather cumbersome.

Example programs written for early computing devices, such as those for Babbage's calculating engine, were naturally presented in "machine language" rather than in a true programming language. For example, the three-address code for Babbage's machine was planned to consist of instructions such as '$V_4 \times V_0 = V_{10}$', where operation signs like '$\times$' would appear on an operation card and subscript numbers like (4, 0, 10) would appear on a separate variable card. The most elaborate program developed by Babbage and Lady Lovelace for this machine was a routine for calculating Bernoulli numbers; see [BA 61, pages 68 and 286–297]. An example Mark I program given in 1946 by Howard Aiken and Grace Hopper (see [RA 73, pages 216–218]) shows that its machine language was considerably more complicated.

Although all of these early programs were in a machine language, it is interesting to note that Babbage had noticed already on 9 July 1836 that machines as well as people could produce programs as output [RA 73, page 349]:

> This day I had for the first time a general but very indistinct conception of the possibility of making an engine work out *algebraic* developments. I mean without *any* reference to the *value* of the letters. My notion is that as the cards (Jacquards) of the Calc. engine direct a series of operations and then recommence with the first so it might perhaps be possible to cause the same cards to punch others equivalent to any given number of repetitions. But there hole [*sic*] might perhaps be small pieces of formulae previously made by the first cards.

In 1914, Leonardo Torres y Quevedo used natural language to describe the steps of a short program for his hypothetical automaton. Helmut Schreyer gave an analogous description in 1939 for the machine that he had helped Konrad Zuse to build (see [RA 73, pages 95–98 and 167]).

To conclude this survey of prior developments, let us take a look at A. M. Turing's famous mathematical paper of 1936 [TU 36], where the concept of a universal computing machine was introduced for theoretical purposes. Turing's machine language was more primitive, not having a built-in arithmetic capability, and he defined a complex program by presenting what amounts to macroexpansions or open subroutines. For example, here was his program for making the machine move to the leftmost 'a' on its working tape:

m-config.	symbol	behavior	final m-config.
$\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, a)$	ə	L	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, a)$
	not ə	L	$\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, a)$
$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, a)$	a		$\mathfrak{C}$
	not a	R	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, a)$
	None	R	$\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, a)$
$\mathfrak{f}_2(\mathfrak{C}, \mathfrak{B}, a)$	a		$\mathfrak{C}$
	not a	R	$\mathfrak{f}_1(\mathfrak{C}, \mathfrak{B}, a)$
	None	R	$\mathfrak{B}$

[In order to carry out this operation, one first sends the machine to state $\mathfrak{f}(\mathfrak{C}, \mathfrak{B}, a)$; it will immediately begin to scan left on the tape (L) until first passing the symbol ə. Then it moves right until either encountering the symbol a or two consecutive blanks; in the first case it enters into state $\mathfrak{C}$ while still scanning the a, and in the second case it enters state $\mathfrak{B}$ after moving to the right of the second blank. Turing used the term "m-configuration" for state.]

Such "skeleton tables," as presented by Turing, represented the highest-level notations for precise algorithm description that had been developed before our story begins — except, perhaps, for Alonzo Church's "λ-notation" [CH 36], which represents an entirely different approach to calculation. Mathematicians would traditionally present the control mechanisms of algorithms informally, and the necessary computations would be expressed by means of equations. There was no concept of assignment (that is, of replacing the value of some variable by a new

value); instead of writing $s \leftarrow -s$ one would write $s_{n+1} = -s_n$, giving a new name to each quantity that would arise during a sequence of calculations.

Zuse's "Plancalculus"

Near the end of World War II, Allied bombs destroyed nearly all of the sophisticated relay computers that Konrad Zuse had been building in Germany since 1936. Only his Z4 machine could be rescued, in what Zuse describes as a hair-raising [*abenteuerlich*] way; and he moved the Z4 to a little shed in a small Alpine village called Hinterstein.

> It was unthinkable to continue practical work on the equipment; my small group of twelve co-workers disbanded. But it was now a satisfactory time to pursue theoretical studies. The Z4 Computer that had been rescued could barely be made to run, and no especially algorithmic language was really necessary to program it anyway. [Conditional commands had consciously been omitted; see [RA 73, page 181].] Thus the PK [*Plankalkül*] arose purely as a piece of desk-work, without regard to whether or not machines suitable for PK's programs would be available in the foreseeable future. [ZU 72, page 6].

Zuse had previously come to grips with the lack of formal notations for algorithms while working on his planned doctoral dissertation [ZU 44]. Here he had independently developed a three-address notation remarkably like that of Babbage. For example, to compute the roots x_1 and x_2 of $x^2 + ax + b = 0$, given $a = V_1$ and $b = V_2$, he prepared the following *Rechenplan* [page 26]:

$$\begin{aligned} V_1 : 2 &= V_3 \\ V_3 \cdot V_3 &= V_4 \\ V_4 - V_2 &= V_5 \\ \sqrt{V_5} &= V_6 \\ V_3(-1) &= V_7 \\ V_7 + V_6 &= V_8 = x_1 \\ V_7 - V_6 &= V_9 = x_2 \end{aligned}$$

He realized that this notation was limited to straight line programs [so-called *starre Pläne*], and he had concluded his previous manuscript with the following remark [ZU 44, page 31]:

> *Unstarre Rechenpläne* constitute the true discipline of higher combinatorial computing; however, they cannot yet be treated in this place.

The completion of this work was the theoretical task Zuse set himself in 1945, and he pursued it very energetically. The result was an amazingly comprehensive language that he called the *Plankalkül* [program calculus], an extension of Hilbert's *Aussagenkalkül* [propositional calculus] and *Prädikatenkalkül* [predicate calculus]. Before laying this project aside, Zuse had completed an extensive manuscript containing programs far more complex than anything written before. Among other things, there were algorithms for sorting into order; for testing the connectivity of a graph represented as a list of edges; for integer arithmetic (including square roots) in binary notation; and for floating-point arithmetic. He even developed algorithms to test whether or not a given logical formula is syntactically well formed, and whether or not such a formula contains redundant parentheses — assuming six levels of precedence between the operators. To top things off, he also included 49 pages of algorithms for playing chess. Who would have believed that such pioneering developments could emerge from the solitary village of Hinterstein? His plans to include algorithms for matrix calculations, series expansions, etc., had to be dropped since the necessary contacts were lacking in that place. Furthermore, his chess playing program treated "en passant captures" incorrectly, because he could find nobody who knew chess any better than he did [ZU 72, pages 32 and 35]!

Zuse's 1945 manuscript unfortunately lay unpublished until 1972, although brief excerpts appeared in 1948 and 1959 [ZU 48, ZU 59]; see also [BW 72], where his work was brought to the attention of English speaking readers for the first time. It is interesting to speculate about what would have happened if he had published everything at once; would many people have been able to understand such radical new ideas?

The monograph [ZU 45] on Plankalkül begins with the following statement of motivation:

> Aufgabe des Plankalküls ist es, beliebige Rechenvorschriften rein formal darzustellen. [The mission of the Plancalculus is to give a purely formal description of any computational procedure.]

So, in particular, the Plankalkül should be able to describe the TPK algorithm; and we had better turn now to this program, before we forget what TPK is all about. Zuse's notation may appear somewhat frightening at first, but we will soon see that it is really not difficult

to understand.

```
01   S2 = (S1.4, SΔ1)
02   P1  |R(V) ⇒ R
03     V |  0     0
04     S |  Δ1    Δ1
05       |√|V| + 5 × V³ ⇒ R
06     V |  0       0     0
07     S |  Δ1      Δ1    Δ1
08   P2  | R(V)   ⇒   R
09     V |  0          0
10     S |11 × Δ1    11 × 2
11       | W2(11) ⌈ R1(V) ⇒ Z
12     V |        | 0  0     0
13     K |        |    i
14     S |        |   Δ1    Δ1
15       |        | Z > 400 ↛ (i, +∞) ⇒ R ⌐(10 − i)
16     V |        | 0                   0 ⌋
17     K |        |
18     S |        |Δ1          1.4      2
19       |        | Z > 400 ↛ (i,  Z) ⇒ R ⌐(10 − i)
20     V |        | 0               0   0 ⌋
21     K |        |
22     S |        |Δ1          1.4  Δ1  2
```

Line *01* of this code is the declaration of a compound data type, and before we discuss the remainder of the program we should stress the richness of data structures provided by Zuse's language (even in its early form [ZU 44]). This feature is, in fact, one of the greatest strengths of the Plankalkül; none of the other languages we shall discuss had such a perceptive notion of data, yet Zuse's proposal was simple and elegant. He started with data of type S0, a single bit [*Ja-Nein-Wert*] whose value is either '−' or '+'. From any given data types $\sigma_0, \ldots, \sigma_{k-1}$, a programmer could define the compound data type $(\sigma_0, \ldots, \sigma_{k-1})$, and individual components of this compound type could be referred to by applying the subscripts $0, \ldots, k-1$ to any variable of that type. Arrays could also be defined by writing $m \times \sigma$, meaning m identical components of type σ; this idea could be repeated, in order to obtain arrays of any desired dimension. Furthermore m could be '□', meaning a list of

variable length, and Zuse made good use of such list structures in his algorithms dealing with graphs, algebraic formulas, and chess play.

Thus the Plankalkül included the important concept of hierarchically structured data, going all the way down to the bit level. Such advanced data structures did not enter again into programming languages until the late 1950s in IBM's Commercial Translator. The idea eventually appeared in many other languages, such as FACT, COBOL, PL/I, and extensions of ALGOL 60; see [CL 61] and [SA 69, page 325].

Binary n-bit numbers in the Plankalkül were represented by type S1.n. Another special type was used for *floating-binary numbers*, namely,

$$\text{S}\Delta\text{1} = (\text{S1.3},\ \text{S1.7},\ \text{S1.22}).$$

The first three-bit component here was for signs and special markers—indicating, for example, whether the number was real or imaginary or zero; the second was for a seven-bit exponent in two's complement notation; and the final 22 bits represented the 23-bit fraction part of a normalized number, with the redundant leading '1' bit suppressed. Thus, for example, the floating point number +400.0 would have appeared as

$$(-+-,\ ---+---,\ ----------------------+--+),$$

and it also could be written

$$(\text{LO},\ \text{LOOO},\ \text{LOOLOOOOOOOOOOOOOOOO}).$$

[The + and − notation has its bits numbered 0, 1, ..., from left to right, while the L and O notation corresponds to binary numbers as we now know them, having their most significant bits at the left.] There was a special representation for "infinite" and "very small" and "undefined" quantities; for example,

$$+\infty = (\text{LLO},\ \text{LOOOO},\ \text{O}).$$

Our TPK program uses $+\infty$ instead of 999 on line *15*, since such a value seems an appropriate way to render the concept 'TOO LARGE'.

Let us return now to the program itself. Line *01* introduces the data type S2, namely, an ordered pair whose first component is a 4-bit integer (type S1.4) and whose second component is floating point (type SΔ1). This data type will be used later for the eleven outputs of the TPK algorithm. Lines *02–07* define the function $f(t)$, and lines *08–22* define the main TPK program.

The hardest thing to get used to about Zuse's notation is the fact that each operation spans several lines; for example, lines *11–14* must

be read as a unit. The second line of each group (labeled 'V') is used to identify the subscripts for quantities named on the top line; thus

$$\begin{array}{lll} \mathrm{R}, & \mathrm{V}, & \mathrm{Z} \\ 0 & 0 & 0 \end{array}$$ stands for the variables R_0, V_0, Z_0.

Operations are done primarily on output variables [*Resultatwerte*] R_k, input variables [*Variablen*] V_k, and intermediate variables [*Zwischenwerte*] Z_k. The 'K' line is used to denote components of a variable, so that, in our example,

$$\begin{array}{l} \mathrm{V} \\ 0 \\ i \end{array}$$ means component i of the input variable V_0.

(A completely blank 'K' line is normally omitted.) Complicated subscripts can be handled by making a zigzag bar from the K line up to the top line, as in line *17* of the program where the notation indicates component $10 - i$ of R_0. The bottom line of each group is labeled A or S, and it is used to specify the type of each variable. Thus the '2' in line *18* of our example means that R_0 is of type S2; the 'Δ1' means that Z_0 is floating point (type SΔ1); and the '1.4' means that i is being represented as a 4-bit binary number (type S1.4).

Zuse remarked [ZU 45, page 10] that the number of possible data types was so large that it would be impossible to indicate a variable's type simply by using typographical conventions as in classical mathematics; thus he realized the importance of apprehending the type of each variable at each point of a program, although such information is usually redundant. This is probably one of the main reasons he introduced the peculiar multiline format. Incidentally, a somewhat similar multiline notation has been used in recent years to describe musical notes [SM 73]; it is interesting to speculate whether musical notation will evolve in the same way that programming languages have.

We are now ready to penetrate further into the meaning of the code above. Each plan begins with a specification part [*Randauszug*], stating the types of all inputs and outputs. Thus, lines *02–04* mean that P1 is a procedure that takes an input V_0 of type SΔ1 (floating point) and produces R_0 of the same type. Lines *08–10* say that P2 maps V_0 of type $11 \times \mathrm{S}\Delta 1$ (namely, a vector of 11 floating-point numbers, the array $a[0{:}10]$ of our TPK algorithm) into a result R_0 of type $11 \times \mathrm{S}2$ (namely, a vector of 11 ordered pairs as described earlier).

The double arrow $\Rightarrow$, which Zuse called the *Ergibt-Zeichen* (yields sign), was introduced for the assignment operation; thus the meaning

of lines *05–07* should be clear. As we have remarked, mathematicians had never used such an operator before; in fact, the systematic use of assignments constitutes a distinct break between computer-science thinking and mathematical thinking. Zuse consciously introduced a new symbol for the new operation, remarking [ZU 45, page 15] that

$$\begin{matrix} \mathrm{Z}+1 \Rightarrow \mathrm{Z} \\ 3 \qquad\quad 3 \end{matrix}$$

was analogous to the equation

$$\begin{matrix} \mathrm{Z} \ +1 = \mathrm{Z} \\ 3.i \qquad\ 3.i+1 \end{matrix}$$

of traditional mathematics. (Incidentally, the publishers of [ZU 48] used the sign $\succcurlyeq$ instead of $\Rightarrow$, but Zuse never actually wrote $\succcurlyeq$ himself.) Notice that the variable receiving a new value appears on the right, while most present-day languages have it on the left. We shall see that there was a gradual "leftist" trend as languages developed.

It remains to understand lines *11–22* of the example. The notation 'W2(n)' represents an iteration, for $i = n-1$ down to 0, inclusive; hence W2(11) stands for the second **for** loop in the TPK algorithm. (The index of such an iteration was always denoted by i, or $i.0$; if another iteration were nested inside, its index would be called $i.1$, etc.) The notation

$$\begin{matrix} \mathrm{R1}(x) \\ 0 \end{matrix}$$

on line *11* stands for the result R_0 of applying procedure P1 to input x. Lines *15–18* of the program mean

$$\textbf{if } \mathrm{Z}_0 > 400 \textbf{ then } \mathrm{R}_0[10-i] := (i, +\infty);$$

notice Zuse's new notation $\dot{\rightarrow}$ for conditionals. Lines *19–22* are similar, with a bar over '$\mathrm{Z}_0 > 400$' to indicate the negation of that relation. There was no equivalent of '**else**' in the Plankalkül, nor were there **go to** statements. Zuse did, however, have the notation 'Fin' with superscripts, to indicate a jump out of a given number of iteration levels and/or to the beginning of a new iteration cycle (see [ZU 72, page 28; ZU 45, page 32]); this idea has recently been revived in the BLISS language [WR 71].

The reader should now be able to understand all of the code given above. In the text accompanying his programs in Plankalkül notation,

Zuse made it a point to state also the mathematical relations between the variables that appeared. He called such a relation an *impliziten Ansatz*; we would now call it an "invariant." This was yet another fundamental idea about programming. Like Zuse's data structures, it disappeared from programming languages during the 1950s, waiting to be enthusiastically received when the time was ripe [HO 71].

Zuse had visions of using the Plankalkül someday as the basis of a programming language that could be translated by machine (see [ZU 72, pages 5, 18, 33, 34]). But in 1945 he was considering first things first — namely, he needed to decide what concepts should be embodied in a notation for programming. We can summarize his accomplishments by saying that the Plankalkül introduced many extremely important ideas, but it lacked the "syntactic sugar" for expressing programs in a readable and easily writable format.

Zuse says he made modest attempts in later years to have the Plankalkül implemented within his own company, "but this project necessarily foundered because the expense of implementing and designing compilers outstripped the resources of my small firm." He also mentions his disappointment that more of the ideas of the Plankalkül were not incorporated into ALGOL 58, since some of ALGOL's original designers knew of his work [ZU 72, page 7]. Such an outcome was probably inevitable, because the Plankalkül was far ahead of its time from the standpoint of available hardware and software development. Most of the other languages we shall discuss started at the other end, by asking what was possible to implement rather than what was possible to write; it naturally took many years for these two approaches to come together and to achieve a suitable synthesis.

Flow Diagrams

On the other side of the Atlantic, Herman H. Goldstine and John von Neumann were wrestling with the same sort of problem that Zuse had faced: How should algorithms be represented in a precise way, at a higher level than the machine's language? Their answer, which was due in large measure to Goldstine's analysis of the problem together with suggestions by von Neumann, Adele Goldstine, and Arthur W. Burks [GO 72, pages 266–268], was quite different from the Plankalkül: They proposed a pictorial representation involving boxes joined by arrows, and they called it a "flow diagram." During 1946 and 1947 they prepared an extensive and carefully worked out treatise on programming based on the idea of flow diagrams [GV 47], and it is interesting to compare their work to that of Zuse. There are striking differences, such as an

emphasis on numerical calculation rather than on data structures; and there also are striking parallels, such as the use of the term "Plan" in the titles of both documents. Although neither work was published in contemporary journals, perhaps the most significant difference was that the treatise of Goldstine and von Neumann was beautifully "varityped" and distributed in quantity to the vast majority of people involved with computers at that time. This fact, coupled with the high quality of presentation and von Neumann's prestige, meant that their report had an enormous impact, forming the foundation for computer programming techniques all over the world. The term "flow diagram" became shortened to "flow chart" and eventually it even became "flowchart" — a word that has entered our language as both noun and verb.

We all know what flowcharts are, but comparatively few people nowadays have seen an authentic original flow diagram. In fact, it is very instructive to go back to the original style of Goldstine and von Neumann, since their inaugural flow diagrams represent a transition point between the mathematical "equality" notation and the computer-science "assignment" operation. Figure 1 shows how the TPK algorithm would probably have looked if Goldstine and von Neumann had been asked to deal with it in 1947.

Several things need to be explained about this original notation. The most important consideration is probably the fact that the boxes containing '$10 \rightarrow i$' and '$i-1 \rightarrow i$' were *not* intended to specify any computation; the viewpoint of Goldstine and von Neumann was significantly different from what we are now accustomed to, and readers will find it worthwhile to ponder this conceptual difference until they are able to understand it. The box '$i-1 \rightarrow i$' represents merely a change in *notation*, as the flow of control passes that point, rather than an action to be performed by the computer. For example, box VII has done the computation necessary to place $2^{-39}(i-1)$ into storage position C.1; so after we pass the box '$i-1 \rightarrow i$' and go through the subsequent junction point to box II, location C.1 now contains $2^{-39}i$. The external notation has changed but location C.1 has not! This distinction between external and internal notations occurs throughout, the external notation being problem-oriented while the actual contents of memory are machine-oriented. The numbers attached to each arrow in the diagram indicate so-called "constancy intervals," where all memory locations have constant contents and all bound variables of the external notation have constant meaning. A "storage table" is attached by a dashed line to the constancy intervals, to show the relevant relations between external and internal values at that point. Thus, for example, we

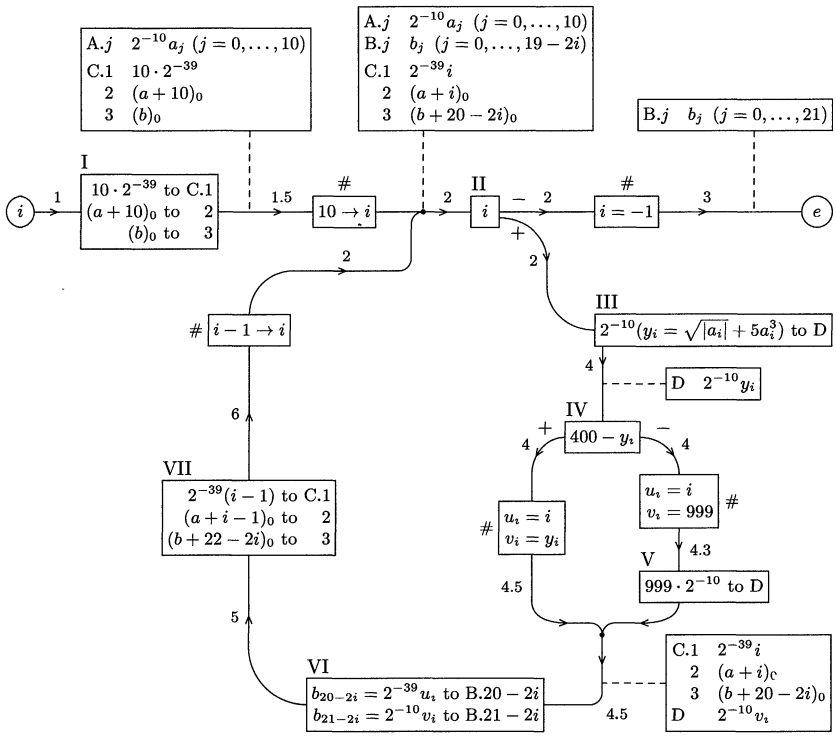


FIGURE 1. A flow diagram in the original Goldstine–von Neumann style.

note that the box '$10 \to i$' does not specify any computation but it provides the appropriate transition from constancy interval 1.5 to constancy interval 2. (See [GV 47, §7.6 and §7.7].)

Such flow diagrams generally contained four kinds of boxes: (a) Operation boxes, marked with a Roman numeral; this is where the computer program was supposed to make appropriate transitions in storage. (b) Alternative boxes, also marked with a Roman numeral, and having two exits marked + and −; this is where the computer control was to branch, depending on the sign of the named quantity. (c) Substitution boxes, marked with a # and using the '$\to$' symbol; this is where the external notation for a bound variable changed, as explained above. (d) Assertion boxes, also marked with a #; this is where important relations between external notations and the current state of the control were specified. The example shows three assertion boxes, one that says '$i = -1$', and two asserting that the outputs u_i and v_i (in a problem-oriented notation) now have certain values. Like substitution boxes,

assertion boxes did not indicate any action by the computer; they merely stated relationships that helped to prove the validity of the program and that might help a programmer to write code for the operation boxes.

The next most prominent feature about original flow diagrams is the fact that programmers were required to be conscious of the *scaling* (that is, the binary point location) of all numbers in the computer memory. A computer word was 40 bits long, and its value was to be regarded as a binary fraction x in the range $-1 \le x < 1$. Thus, for example, the flowchart in Figure 1 assumes that $2^{-10}a_j$ is initially present in storage position A.j, not the value a_j itself; the outputs b_j are similarly scaled.

The final mystery that needs to be revealed is the meaning of notations such as $(a+i)_0$, $(b)_0$, etc. In general, 'x_0' was used when x was an integer machine address; it represented the number $2^{-19}x + 2^{-39}x$, namely, a binary word with x appearing twice, in bit positions 9 to 20 and 29 to 40 (counting from the left). Such a number could be used in their machine to modify the addresses of 20-bit instructions that appeared in either half of a 40-bit word.

Once a flow diagram such as this had been drawn up, the remaining task was to prepare so-called "static coding" for boxes marked with Roman numerals. In this task programmers would use their problem-solving ability, together with their knowledge of machine language and the information from storage tables and assertion boxes, to make the required transitions. For example, in box VI one should use the facts that $u_i = i$, that storage D contains $2^{-10}v_i$, that storage C.1 contains $2^{-39}i$, and that storage C.3 contains $(b + 20 - 2i)_0$ (a word corresponding to the location of variable B.$20-2i$) to carry out the specified assignments. The job of box VII is slightly trickier: One of the tasks, for example, is to store $(b + 22 - 2i)_0$ in location C.3; the programmer was supposed to resolve this by adding $2 \cdot (2^{-19} + 2^{-39})$ to the previous contents of C.3. In general, the job of static coding required a fairly high level of problem-solving ability, and it was far beyond the state of the art in those days to get a computer to do such a thing. As with the Plankalkül, the notation needed to be simplified if it was to be suitable for machine implementation.

Let us make one final note about flow diagrams in their original form: Goldstine and von Neumann did not suggest any notation for subroutine calls, hence the function $f(t)$ in the TPK algorithm has been written in-line. In [GV 47, §12] there is a flow diagram for the algorithm that a loading routine must follow in order to relocate subroutines from a library, but there is no example of a flow diagram for a driver program that calls a subroutine. An appropriate extension of flow diagrams to

subroutine calls could surely be made, but such a change would have made our example less "authentic."

A Logician's Approach

Let us now turn to the proposals made by Haskell B. Curry, who was working at the Naval Ordnance Laboratory in Silver Spring, Maryland; his activity was partly contemporaneous with that of Goldstine and von Neumann, since the last portion of [GV 47] was not distributed until 1948.

Curry wrote two lengthy memoranda [CU 48, CU 50] that have never been published; the only appearance of his work in the open literature has been the brief and somewhat cryptic summary in [CU 50′]. He had prepared a rather complex program for ENIAC in 1946, and this experience led him to suggest a notation for program construction that is more compact than flowcharts.

His aims, which correspond to important aspects of what we now call "structured programming," were quite laudable [CU 50, paragraph 34]:

> The first step in planning the program is to analyze the computation into certain main parts, called here divisions, such that the program can be synthesized from them. Those main parts must be such that they, or at any rate some of them, are independent computations in their own right, or are modifications of such computations.

But in practice his proposal was not especially successful, because the way he factored a problem was not very natural. His components tended to have several entrances and several exits, and perhaps his mathematical abilities tempted him too strongly to pursue the complexities of fitting such pieces together. As a result, the notation he developed was somewhat eccentric, and the work was left unfinished. Here is how he might have represented the TPK algorithm:

$$
\begin{aligned}
\mathrm{F}(t) &= \{\sqrt{|t|} + 5t^3 : \mathrm{A}\} \\
\mathrm{I} &= \{10 : i\} \to \{t = \mathrm{L}(a+i)\} \to \mathrm{F}(t) \to \{\mathrm{A} : y\} \\
&\quad \to \mathrm{II} \to \mathrm{It}_7(0, i) \to \mathrm{O}_1 \,\&\, \mathrm{I}_2 \\
\mathrm{II} &= \{x = \mathrm{L}(b + 20 - 2i)\} \to \{i : x\} \to \mathrm{III} \\
&\quad \to \{w = \mathrm{L}(b + 21 - 2i)\} \to \{y : w\} \\
\mathrm{III} &= \{y > 400\} \to \{999 : y\} \,\&\, \mathrm{O}_1
\end{aligned}
$$

The following explanations should suffice to make the example clear, although they do not reveal the full generality of his language:

$\{E : x\}$ means "compute the value of expression E and store it in location x."

A denotes the accumulator of the machine.

$\{x = L(E)\}$ means "compute the value of expression E as a machine location and substitute it for all appearances of 'x' in the following instruction groups."

$X \rightarrow Y$ means "substitute instruction group Y for the first exit of instruction group X."

I_j denotes the jth entrance of this routine, namely, the beginning of its jth instruction group.

O_j denotes the jth exit of this routine (he used the words "input" and "output" for entrance and exit).

$\{x > y\} \rightarrow O_1 \,\&\, O_2$ means "if $x > y$, go to O_1, otherwise to O_2."

$It_7(m, i) \rightarrow O_1 \,\&\, O_2$ means "decrease i by 1, then if $i \geq m$ go to O_2, otherwise to O_1." (Here "It" stands for "iterate.")

Actually the main feature of interest in Curry's early work is not this programming language, but rather the algorithms he discussed for converting parts of it into machine language. He gave a recursive description of a procedure to convert fairly general arithmetic expressions into code for a one-address computer, thereby being the first person to describe the code-generation phase of a compiler. (Syntactic analysis was not specified; he gave recursive reduction rules analogous to well-known constructions in mathematical logic, assuming that any formula could be parsed properly.) His motivation for doing this was stated in [CU 50′, page 100]:

> Now von Neumann and Goldstine have pointed out that, as programs are made up at present, we should not use the technique of program composition to make the simpler sorts of programs — these would be programmed directly — but only to avoid repetitions in forming programs of some complexity. Nevertheless, there are three reasons for pushing clear back to formation of the simplest programs from the basic programs [that is, from machine language instructions], viz.: (1) Experience in logic and in mathematics shows that an insight into principles is often best obtained by a consideration of cases too simple for practical use — e.g., one gets an insight into the nature of a group by considering the permutations of three letters, etc.... (2) It is quite possible that the technique of program composition can completely replace the elaborate methods of Goldstine and von

> Neumann; while this may not work out, the possibility is at least worth considering. (3) The technique of program composition can be mechanized; if it should prove desirable to set up programs, or at any rate certain kinds of them, by machinery, presumably this may be done by analyzing them clear down to the basic programs.

The program that his algorithm would have constructed for $\mathrm{F}(t)$, if the expression t^3 were replaced by $t \cdot t \cdot t$, is

$$\{|t| : \mathrm{A}\} \to \{\sqrt{\mathrm{A}} : \mathrm{A}\} \to \{\mathrm{A} : w\} \to \{t : \mathrm{R}\} \to \{t\mathrm{R} : \mathrm{A}\} \to \{\mathrm{A} : \mathrm{R}\}$$
$$\to \{t\mathrm{R} : \mathrm{A}\} \to \{\mathrm{A} : \mathrm{R}\} \to \{5\mathrm{R} : \mathrm{A}\} \to \{\mathrm{A} + w : \mathrm{A}\}.$$

Here w is a temporary storage location, and R is a register used in multiplication.

An Algebraic Interpreter

The three languages we have seen so far were never implemented; they served purely as conceptual aids during the programming process. Such conceptual aids were obviously important, but they still left the programmer with a lot of mechanical things to do, and there were many chances for errors to creep in.

The first "high-level" programming language actually to be implemented was the Short Code, originally suggested by John W. Mauchly in 1949. William F. Schmitt coded it for the BINAC at that time. Later in 1950, Schmitt recoded Short Code for the UNIVAC, with the assistance of Albert B. Tonik, and J. Robert Logan revised the program in January 1952. Details of the system have never been published, and the earliest extant programmer's manual [RR 55] seems to have been written originally in 1952.

The absence of data about the early Short Code indicates that it was not an instant success, in spite of its eventual historic significance. This lack of popularity is not surprising when we consider the small number of scientific users of UNIVAC equipment in those days; in fact, the most surprising thing is that an algebraic language such as this was not developed first at the mathematically oriented centers of computer activity. Perhaps the reason is that mathematicians were so conscious of efficiency considerations that they could not imagine wasting any extra computer time for something that a programmer could do by hand. Mauchly had greater foresight in this regard; and J. R. Logan put it this way [RR 55]:

By means of the Short Code, any mathematical equations may be evaluated by the mere expedient of writing them down. There is a simple symbological transformation of the equations into code as explained by the accompanying write-up. The need for special programming has been eliminated.

In our comparisons of computer time with respect to time consumed by manual methods, we have found so far a speed ratio of at least fifty to one. We expect better results from future operations.

... It is expected that future use of the Short Code will demonstrate its power as a tool in mathematical research and as a checking device for some large-scale problems.

We cannot be certain how UNIVAC Short Code looked in 1950, but it probably was closely approximated by the 1952 version, when TPK could have been coded in the following way:

Memory equivalents: $i \equiv$ `W0`, $t \equiv$ `T0`, $y \equiv$ `Y0`.

Eleven inputs go respectively into words `U0`, `T9`, `T8`, ..., `T0`.

Constants:

`Z0` =	`000000000000`	
`Z1` =	`010000000051`	[1.0 in floating-decimal form]
`Z2` =	`010000000052`	[10.0]
`Z3` =	`040000000053`	[400.0]
`Z4` =	`ΔΔΔTOOΔLARGE`	['TOO LARGE']
`Z5` =	`050000000051`	[5.0]

Equation number recall information [labels]:

0 = line *01*, 1 = line *06*, 2 = line *07*

Short code:

	Equations	*Coded representation*
00	$i = 10$	`00 00 00 W0 03 Z2`
01	0: $y = (\sqrt{} \text{ abs } t) + 5 \text{ cube } t$	`T0 02 07 Z5 11 T0`
02		`00 Y0 03 09 20 06`
03	y 400 if≤to 1	`00 00 00 Y0 Z3 41`
04	i print, 'TOO LARGE' print-and-return	`00 00 Z4 59 W0 58`
05	0 0 if=to 2	`00 00 00 Z0 Z0 72`
06	1: i print, y print-and-return	`00 00 Y0 59 W0 58`
07	2: `T0 U0` shift	`00 00 00 T0 U0 99`
08	$i = i - 1$	`00 W0 03 W0 01 Z1`
09	0 i if≤to 0	`00 00 00 Z0 W0 40`
10	stop	`00 00 00 00 ZZ 08`

Each UNIVAC word consisted of twelve 6-bit bytes, and the Short Code equations were "symbologically" transliterated into groups of six 2-byte packets using the following equivalents (among others):

01 −	06 abs	1*n* (*n*+2)nd power	59 print and return carriage
02)	07 +	2*n* (*n*+2)nd root	7*n* if=to *n*
03 =	08 pause	4*n* if≤to *n*	99 cyclic shift of memory
04 /	09 (	58 print and tab	S*n*, T*n*, ..., Z*n* quantities

Thus, '$i = 10$' would be encoded as the word '00 00 00 W0 03 Z2' shown; packets of 00s could be used at the left to fill a word. Multiplication was indicated simply by juxtaposition (see line *01*).

The system was an *algebraic interpreter*, namely, an interpretive routine that repeatedly scanned the coded representation and performed the appropriate operations. The interpreter processed each word from right to left, so that it would see the '=' sign of an assignment operation last. This fact needed to be understood by the programmer, who had to break long equations appropriately into several words (see lines *01* and *02*). See also the print instructions on lines *04* and *06*, where the codes run from right to left.

This explanation should suffice to explain the TPK program above, except for the 'shift' on line *07*. Short Code had no provision for subscripted variables, but it did have a 99 order, which performed a cyclic shift in a specified block of memory. For example, line *07* of the example program means

$$temp = \mathtt{T0}, \quad \mathtt{T0} = \mathtt{T1}, \quad \ldots, \quad \mathtt{T9} = \mathtt{U0}, \quad \mathtt{U0} = temp;$$

and fortunately this facility is all that the TPK algorithm needs.

The following press release from Remington Rand appeared in the *Journal of the Association for Computing Machinery* **2** (1955), page 291:

> Automatic programming, tried and tested since 1950, eliminates communication with the computer in special code or language. ... The Short-Order Code is in effect an engineering "electronic dictionary" ... an interpretive routine designed for the solution of one-shot mathematical and engineering problems.

(Several other automatic programming systems, including "B-zero" — which we shall discuss later — were also announced at that time.) This is one of the few places where Short Code has been mentioned in the open literature. Grace Hopper referred to it briefly in [HO 52, page 243] (calling it "short-order code"), [HO 53, page 142] ("short-code"), and

[HO 58, page 165] ("Short Code"). In [HM 53, page 1252] it is stated that the "short code" system was "only a first approximation to the complete plan as originally conceived." This is probably true, but several discrepancies between [HM 53] and [RR 55] indicate that the authors of [HM 53] were not fully familiar with UNIVAC Short Code as it actually existed.

The Intermediate PL of Burks

Independent efforts to simplify the job of coding were being made at this time by Arthur W. Burks and his colleagues at the University of Michigan. The overall goal of their activities was to investigate the process of going from the vague "ordinary business English" description of some data-processing problem to the "internal program language" description of a machine-language program for that problem; and, in particular, to break this process up into a sequence of smaller steps [BU 51, page 12]:

> This has two principal advantages. First, smaller steps can more easily be mechanized than larger ones. Second, different kinds of work can be allocated to different stages of the process and to different specialists.

In 1950, Burks sketched a so-called "intermediate program language" that was to be the step one notch above the internal program language. Instead of spelling out complete rules for this intermediate language, he took portions of two machine programs previously published in [BU 50] and showed how they could be expressed at a higher level of abstraction. From these two examples it is possible to make a reasonable guess at how he might have written the TPK algorithm at that time:

1. $10 \to i$	
To 10	
From 1,35	
10. $A + i \to 11$	Compute location of a_i
11. $[A + i] \to t$	Look up a_i and transfer to storage
12. $\lvert t\rvert^{1/2} + 5t^3 \to y$	$y_i = \sqrt{\lvert a_i\rvert} + 5a_i^3$
13. $400, y;\ 20, 30$	Determine if $v_i = y_i$
To 20 if $y > 400$	
To 30 if $y \leq 400$	
From 13	
20. $999 \to y$	$v_i = 999$
To 30	

From 13, 20

30. $(B + 20 - 2i)' \rightarrow 31$	Compute location of b_{20-2i}
31. $i \rightarrow [B + 20 - 2i]$	$b_{20-2i} = i$
32. $(B + 20 - 2i) + 1 \rightarrow 33$	Compute location of b_{21-2i}
33. $y \rightarrow [(B + 20 - 2i) + 1]$	$b_{21-2i} = v_i$
34. $i - 1 \rightarrow i$	$i \rightarrow i + 1$
35. $i, 0;\ 40, 10$	Repeat cycle until i negative

To 40 if $i < 0$
To 10 if $i \geq 0$

From 35

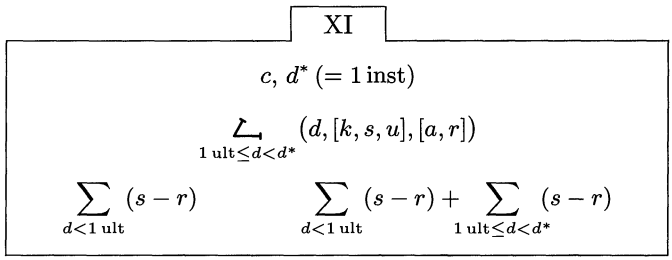
40. F	Stop execution

Comments at the right of this program attempt to imitate Burks's style of writing comments at that time; they succeed in making the program almost completely self-explanatory. Notice that the assignment operation is well established by now; Burks used it also in the somewhat unusual form '$i \rightarrow i+1$' shown in the comment to instruction 34 [BU 50, page 41].

The prime symbol that appears within instruction 30 meant that the computer was to save this intermediate result, as it was a common subexpression that could be used without recomputation. Burks mentioned that several of the ideas embodied in this language were due to Janet Wahr, Don Warren, and Jesse Wright. He also made some comments about the feasibility of automating the process [BU 51, page 13]:

> Methods of assigning addresses and of expanding abbreviated commands into sequences of commands can be worked out in advance. Hence the computer could be instructed to do this work. ... It should be emphasized, however, that even if it were not efficient to use a computer to make the translation, the Intermediate PL would nevertheless be useful to the human programmer in planning and constructing programs.

At the other end of the spectrum, nearer to ordinary business language, Burks and his colleagues later proposed an abstract form of description that may be of independent interest, even though it does not relate to the rest of our story. The accompanying example suffices to give the flavor of their "first abstraction language," proposed in 1954. On the first line, c denotes the customer's name and address, and d^* is '1 inst', the first of the current month. The symbol $\angle_i(x_1, \ldots, x_n)$ was used to denote a list of all n-tuples $(x_1, \ldots, x_n)$ of category i, sorted into increasing order by the first component x_1, and the meaning of the



Form XI: Customer's Statement

second line is "a listing, in order of the date d, of all invoices and all remittances for the past month." Here $[k, s, u]$ was an invoice, characterized by its number k, its dollar amount s, and its discount u; similarly, $[a, r]$ was a remittance of r dollars, identified by number a; and '1 ult' means the first of the previous month. The bottom gives the customer's old balance from the previous statement, and the new balance on the right. "The notation is so designed as to leave unprejudiced the method of the statement's preparation" [BC 54]. Such notations have not won over the business community, however, perhaps for reasons explained by Grace Hopper in [HO 58, page 198]:

> I used to be a mathematics professor. At that time I found there were a certain number of students who could not learn mathematics. I then was charged with the job of making it easy for businessmen to use our computers. I found it was not a question of whether they could learn mathematics or not, but whether they would. ... They said, "Throw those symbols out—I do not know what they mean, I have not time to learn symbols." I suggest a reply to those who would like data processing people to use mathematical symbols that they make them first attempt to teach those symbols to vice-presidents or a colonel or admiral. I assure you that I tried it.

Rutishauser's Contribution

Now let us shift our attention once again to Europe, where the first published report on methods for machine code generation was about to appear. Heinz Rutishauser was working with the Z4 computer, which by then had been rebuilt and moved to the Swiss Federal Institute of Technology [Eidgenössische Technische Hochschule (ETH)] in Zurich; and plans were afoot to build a brand new machine there. The background

of Rutishauser's contribution can best be explained by quoting from a letter he wrote some years later [RU 63]:

> I am proud that you are taking the trouble to dig into my 1952 paper. On the other hand it makes me sad, because it reminds me of the premature death of an activity that I had started hopefully in 1949, but could not continue after 1951 because I had to do other work — to run practically singlehanded a fortunately slow computer as mathematical analyst, programmer, operator and even troubleshooter (but not as an engineer). This activity forced me also to develop new numerical methods, simply because the ones then known did not work in larger problems. Afterwards when I would have had more time, I did not come back to automatic programming but found more taste in numerical analysis. Only much later I was invited — more for historical reasons, as a living fossil so to speak, than for actual capacity — to join the ALGOL venture. The 1952 paper simply reflects the stage where I had to give up automatic programming, and I was even glad that I was able to put out that interim report (although I knew that it was final).

Rutishauser's comprehensive treatise [RU 52] described a hypothetical computer and a simple algebraic language, together with complete flowcharts for two compilers for that language. One compiler expanded all loops completely, while the other produced compact code using index registers. His source language was somewhat restrictive, since there was only one nonsequential control structure (the **for** statement); but that control structure was in itself an important contribution to the later development of programming languages. Here is how he might have written the TPK algorithm:

01 Für $i = 10(-1)0$
02 $a_i \Rightarrow t$
03 $(\text{Sqrt Abs } t) + (5 \times t \times t \times t) \Rightarrow y$
04 $\text{Max}(\text{Sgn}(y - 400), 0) \Rightarrow h$
05 $\text{Z } 0_i \Rightarrow b_{20-2i}$
06 $(h \times 999) + ((1 - h) \times y) \Rightarrow b_{21-2i}$
07 Ende Index i
08 Schluss

Since no '**if** ... **then**' construction — much less **go to** — was present in Rutishauser's language, the computation of "y if $y \leq 400$, or 999 if $y > 400$" has been done here in terms of the Max and Sgn functions

he did have, plus appropriate arithmetic; see lines *04* and *06*. (The function Sgn(x) is 0 if $x = 0$, or $+1$ if $x > 0$, or -1 if $x < 0$.) Another problem was that he gave no easy mechanism for converting between indices and other variables; indices (that is, subscripts) were completely tied to Für–Ende loops. Our program therefore invokes a trick to get i into the main formula on line *04*: 'Z 0_i' is intended to use the Z instruction, which transferred an indexed address to the accumulator in Rutishauser's machine [RU 52, page 10], and we have used that instruction in such a way that his compiler would produce the correct code. It is not clear whether or not he would have approved of this trick; if not, we could have introduced another variable, maintaining its value equal to i. But since he wrote a paper several years later [RU 61] entitled "Interference with an ALGOL procedure," there is some reason to believe he would have enjoyed the trick very much.

As with Short Code, the algebraic source code symbols had to be transliterated before the program was amenable to computer input, and the programmer had to allocate storage locations for the variables and constants. Here is how our TPK program would have been converted to a sequence of (floating-point) numbers on punched paper tape, using the memory assignments $a_i \equiv 100 + i$, $b_i \equiv 200 + i$, $0 \equiv 300$, $1 \equiv 301$, $5 \equiv 302$, $400 \equiv 303$, $999 \equiv 304$, $y \equiv 305$, $h \equiv 306$, $t \equiv 307$:

```
        Für   i   = 10  (−1)  0
01      10^12 , 50 ,   10 ,  −1 ,  0 , Q ,
        begin stmt   a   sub i   ≥≥   t
02       010000 ,  100 , .001 , 200000 , 307 , Q ,
        begin stmt    (     t     Abs   dummy   Sqrt
03       010000 ,  010000 , 307 , 110000 ,   0 ,   350800 ,
        dummy    )     +     (     5     ×     t
          0 ,   200000 , 020000 , 010000 , 302 , 060000 , 307 ,
          ×     t     ×     t     )     ≥≥    y
        060000 , 307 , 060000 , 307 , 200000 , 200000 , 305 , Q ,
        begin stmt    (     (     y     −     400     )
04       010000 ,  010000 , 010000 , 305 , 030000 , 303 , 200000 ,
          Sgn   dummy    )     Max   0     ≥≥    h
        100000 ,   0 ,   200000 , 080000 , 300 , 200000 , 306 , Q ,
        begin stmt      Z    0  sub i   ≥≥   b20  sub −2i
05       010000 ,  0 , 230000 , 0 , .001 , 200000 , 220 ,  −.002 ,  Q ,
        begin stmt    (     h     ×     999     )     +
06       010000 ,  010000 , 306 , 060000 , 304 , 200000 , 020000 ,
```

((1 − h) × y
010000, 010000, 301, 030000, 306, 200000, 060000, 305,
) ≻≺ b_{21} sub $-2i$
200000, 200000, 221, −.002, Q,
Ende
07 Q, Q,
Schluss
08 Q, Q.

Here Q represents a special flag that was distinguishable from all numbers. The transliteration is straightforward, except that unary operators such as 'Abs x' have to be converted to binary operators 'x Abs 0'. An extra left parenthesis is inserted before each formula, to match the ≻≺ (which has the same code as right parenthesis). Subscripted variables whose address is $\alpha + \Sigma c_j i_j$ are specified by writing the base address α followed by a sequence of values $c_j 10^{-3j}$; this scheme allows multiple subscripts to be treated in a simple way. The operator codes were chosen to make life easy for the compiler; for example, 020000 was the machine operation 'add' as well as the input code for +, so the compiler could treat almost all operations alike. The codes for left and right parentheses were the same as the machine operations to load and store the accumulator, respectively.

Since his compilation algorithm is published and reasonably simple, we can exhibit exactly the object code that would be generated from the source input above. The output is fairly long, but we shall consider it in its entirety in view of its importance from the standpoint of compiler history. Each word in Rutishauser's machine held two instructions, and there were 12 decimal digits per instruction word. The machine's accumulator was called Op.

Machine instructions	Symbolic form
230010 200050	$10 \to \text{Op}$, $\text{Op} \to i$,
230001 120000	$1 \to \text{Op}$, $-\text{Op} \to \text{Op}$,
200051 230000	$\text{Op} \to i'$, $0 \to \text{Op}$
200052 220009	$\text{Op} \to i''$, $* + 1 \to \text{IR}_9$
239001 200081	$1 + \text{IR}_9 \to \text{Op}$, $\text{Op} \to \text{L}_1$
000000 230100	no-op, loc $a_0 \to \text{Op}$
200099 010050	$\text{Op} \to \text{T}$, $i \to \text{Op}$
020099 210001	$\text{Op} + \text{T} \to \text{Op}$, $\text{Op} \to \text{IR}_1$
011000 200307	$(\text{IR}_1) \to \text{Op}$, $\text{Op} \to t$
010307 110000	$t \to \text{Op}$, $\lvert\text{Op}\rvert \to \text{Op}$
220009 350800	$* + 1 \to \text{IR}_9$, go to Sqrt
000000 000000	no-op, no-op

```
200999 010302    Op → P1,   5 → Op
060307 060307    Op × t → Op,   Op × t → Op
060307 200998    Op × t → Op,   Op → P2
010999 020998    P1 → Op,   Op + P2 → Op
200305 010305    Op → y,   y → Op
030303 200999    Op − 400 → Op,   Op → P1
010999 100000    P1 → Op,   Sgn Op → Op
200998 010998    Op → P2,   P2 → Op
080300 200306    Max(Op, 0) → Op,   Op → h,
230000 200099    0 → Op,   Op → T
010050 020099    i → Op,   Op + T → Op
210001 230220    Op → IR1,   loc b20 → Op
200099 230002    Op → T,   2 → Op
120000 060050    −Op → Op,   Op × i → Op
020099 210002    Op + T → Op,   Op → IR2
010000 231000    (0) → Op,   IR1 → Op
202000 230221    Op → (IR2),   loc b21 → Op
200099 230002    Op → T,   2 → Op
120000 060050    −Op → Op,   Op × i → Op
020099 210001    Op + T → Op,   Op → IR1
010301 030306    1 → Op,   Op − h → Op
200999 010306    Op → P1,   h → Op
060304 200998    Op × 999 → Op,   Op → P2
010999 060305    P1 → Op,   Op × y → Op
200997 010998    Op → P3,   P2 → Op
020997 201000    Op + P3 → Op,   Op → (IR1)
010081 210009    L1 → Op,   Op → IR9
010050 220008    i → Op,   ∗ + 1 → IR8
030052 388003    Op − i'' → Op,   to (IR8 + 3) if Op = 0
010050 020051    i → Op,   Op + i' → Op
200050 359000    Op → i,   go to (IR9)
000000 999999    no-op,   stop
999999           stop
```

(Several bugs on pages 39–40 of [RU 52] need to be corrected in order to produce this code, but Rutishauser's original intent is reasonably clear. The most common error made by a person who first tries to write a compiler is to confuse compilation time with object-code time, and Rutishauser gets the honor of being first to make this error!)

The code above has the interesting property that it is completely relocatable — even if we move all instructions up or down by half a word. Careful study of the output shows that index registers were treated rather awkwardly; but after all, this was 1951, and many compilers even nowadays produce far more disgraceful code than this.

Rutishauser published slight extensions of his source language notation in [RU 55] and [RU 55′].

Böhm's Compiler

An Italian graduate student, Corrado Böhm, developed a compiler at the same time and in the same place as Rutishauser, so it is natural to assume — as many people have — that they worked together. But in fact, their methods had essentially nothing in common. Böhm (who was a student of Eduard Stiefel) developed a language, a machine, and a translation method of his own, during the latter part of 1950, knowing only of [GV 47] and [ZU 48]; he learned of Rutishauser's similar interests only after he had submitted his doctoral dissertation in 1951, and he amended the dissertation at that time in order to clarify the differences between their approaches.

Böhm's dissertation [BO 52] was especially remarkable because he not only described a complete compiler, he also defined that compiler in its own language! And the language was interesting in itself, because *every* statement (including input statements, output statements, and control statements) was a special case of an assignment statement. Here is how TPK looks in Böhm's language:

A. Set $i = 0$ (plus the base address 100 for the input array a).
$$\pi' \to A$$
$$100 \to i$$
$$B \to \pi$$

B. Let a new input a_i be given. Increase i by unity, and proceed to C if $i > 10$, otherwise repeat B.
$$\pi' \to B$$
$$? \to \downarrow i$$
$$i + 1 \to i$$
$$[(1 \cap (i \dot{-} 110)) \cdot C] + [(1 \dot{-} (i \dot{-} 110)) \cdot B] \to \pi$$

C. Set $i = 10$.
$$\pi' \to C$$
$$110 \to i$$

D. Call x the number a_i, and prepare to calculate its square root r (using subroutine R), returning to E.
$$\pi' \to D$$
$$\downarrow i \to x$$
$$E \to X$$
$$R \to \pi$$

E. Calculate $f(a_i)$ and attribute it to y. If $y > 400$, continue at F, otherwise at G.
$$\pi' \to E$$
$$r + 5 \cdot \downarrow i \cdot \downarrow i \cdot \downarrow i \to y$$
$$[(1 \cap (y \dot{-} 400)) \cdot F] + [(1 \dot{-} (y \dot{-} 400)) \cdot G] \to \pi$$

F. Output the actual value of i, then the value 999 ("too large"). Proceed to H.	$\pi' \to F$ $i \dot{-} 100 \to ?$ $999 \to ?$ $H \to \pi$
G. Output the actual values of i and y.	$\pi' \to G$ $i \dot{-} 100 \to ?$ $y \to ?$ $H \to \pi$
H. Decrease i by unity, and return to D if $i \geq 0$. Otherwise stop.	$\pi' \to H$ $i \dot{-} 1 \to i$ $[(1 \dot{-} (100 \dot{-} i)) \cdot D] + [(1 \cap (100 \dot{-} i)) \cdot \Omega] \to \pi$

Comments in an approximation to Böhm's style appear here on the left while the program itself is on the right. As remarked earlier, everything in Böhm's language appears as an assignment. The statement '$B \to \pi$' means "go to B," that is, set the program counter π to the value of variable B. The statement '$\pi' \to B$' means "this is label B"; a loading routine preprocesses the object code, using such statements to set the initial value of variables like B rather than to store an instruction in memory. The symbol '?' stands for the external world; hence the statement '$? \to x$' means "input a value and assign it to x", and the statement '$x \to ?$' means "output the current value of x." An arrow '$\downarrow$' is used to indicate indirect addressing (restricted to one level); thus, '$? \to \downarrow i$' in part B means "read one input into the location whose value is i," namely, into a_i.

Böhm's machine operated only on *nonnegative integers* of 14 decimal digits. As a consequence, his operation $x \dot{-} y$ was the logician's "saturating subtraction,"

$$x \dot{-} y = \begin{cases} x - y, & \text{if } x > y; \\ 0, & \text{if } x \leq y. \end{cases}$$

He also used the notation $x \cap y$ for $\min(x, y)$. Thus it can be verified that

$$1 \cap (i \dot{-} j) = \begin{cases} 1, & \text{if } i > j; \\ 0, & \text{if } i \leq j; \end{cases}$$

$$1 \dot{-} (i \dot{-} j) = \begin{cases} 0, & \text{if } i > j; \\ 1, & \text{if } i \leq j. \end{cases}$$

Because of these identities, the complicated formula at the end of part B is equivalent to a conditional branch,

$$C \to \pi \qquad \text{if } i > 110;$$
$$B \to \pi \qquad \text{if } i \le 110.$$

It is easy to read Böhm's program with these notational conventions in mind. Notice that part C doesn't end with '$D \to \pi$', although it could have; similarly we could have deleted '$B \to \pi$' after part A. (Böhm omitted a redundant go-to statement only once, out of six chances that he had in [BO 52].)

Part D shows how subroutines are readily handled in his language, although he did not explicitly mention them. The integer square root subroutine can be programmed as follows, given the input x and the exit location X:

R. Set $r = 0$ and $t = 2^{46}$.	$\pi' \to R$
	$0 \to r$
	$70368744177664 \to t$
	$S \to \pi$
S. If $r + t \le x$, go to T,	$\pi' \to S$
otherwise go to U.	$r + t \dot{-} x \to u$
	$[(1 \dot{-} u) \cdot T] + [(1 \cap u) \cdot U] \to \pi$
T. Decrease x by $r + t$,	$\pi' \to T$
divide r by 2,	$x \dot{-} r \dot{-} t \to x$
increase r by t,	$r : 2 + t \to r$
and go to V.	$V \to \pi$
U. Divide r by 2.	$\pi' \to U$
	$r : 2 \to r$
V. Divide t by 4.	$\pi' \to V$
If $t = 0$, exit to X,	$t : 4 \to t$
otherwise return to S.	$[(1 \dot{-} t) \cdot X] + [(1 \cap t) \cdot S] \to \pi$

(This algorithm is equivalent to the classical pencil-and-paper method for square roots, adapted to binary notation. It was given in hardware-oriented form as example P9.18 by Zuse in [ZU 45, pages 143–159]. To prove its validity, one can verify that the following invariant relations hold when we reach part S:

t is a power of 4;
r is a multiple of $4t$;
$r^2/4t + x$ is the initial value of x;
$0 \le x < 2r + 4t$.

At the conclusion of the algorithm these conditions hold with $t = 1/4$; so r is the integer square root and x is the remainder.)

Böhm's one-pass compiler was capable of generating instructions rapidly, as the input was being read from paper tape. Unlike Rutishauser, Böhm recognized operator precedence in his language; for example, $r : 2 + t$ was interpreted as $(r : 2) + t$, because the division operator ':' took precedence over addition. However, Böhm did not allow parentheses to be mixed with precedence relations: If an expression began with a left parenthesis, the expression had to be *fully* parenthesized even when associative operators were present; on the other hand, if an expression did *not* begin with a left parenthesis, precedence was considered but no parentheses were allowed within it. The complete program for his compiler consisted of 114 assignments, broken down as follows:

i) 59 statements to handle formulas with parentheses:
ii) 51 statements to handle formulas with operator precedence;
iii) 4 statements to decide between (i) and (ii).

There was also a loading routine, described by 16 assignment statements. So the compiler amounted to only 130 statements in all, including 33 statements that were merely labels ($\pi' \rightarrow \cdots$). Such brevity is especially surprising when we realize that a good deal of the program was devoted solely to checking the input for correct syntax; this check was not complete, however. (It appears to be necessary to add one more statement in order to fix a bug in his program, caused by overlaying information when a left parenthesis follows an operator symbol; but even with this "patch" the compiler is quite elegant.)

Rutishauser's parsing technique often required order n^2 steps to process a formula of length n. His idea was to find the leftmost pair of parentheses that has the highest level, so that they enclose a parenthesis-free formula α, and to compile the code for '$\alpha \rightarrow \mathrm{P}_q$'; then the subformula '$(\alpha)$' was simply replaced by 'P_q', q was increased by 1, and the process was iterated until no parentheses remained. Böhm's parsing technique, on the other hand, was of order n, generating instructions in what amounts to a linked binary tree, while the formula was being read in. To some extent, his algorithm anticipated modern list-processing techniques, which were first made explicit by Newell, Shaw, and Simon about 1956 (see [KN 68, §2.6]).

The table on the following page indicates briefly how Böhm's algorithm would have translated the statement $((a : (b \cdot c)) + ((d \cap e) \div f)) \rightarrow g$, assuming that the bug referred to above had been removed. After the operations shown in the table, the contents of the tree would be punched

Input	Current partial instruction	Current position in tree	Contents of tree (instructions and stack pointers) 1	2	3	4	5
(		①	⓪				
(		②	⓪	①			
a	a	②	⓪	①			
:	$a:$	②	⓪	①			
(		③	⓪	$a:$ ③, ①	②		
b	b	③	⓪	$a:$ ③, ①	②		
·	$b\cdot$	③	⓪	$a:$ ③, ①	②		
c	$b\cdot c$	③	⓪	$a:$ ③, ①	②		
)	③	②	⓪	$a:$ ③, ①	$b\cdot c\to$ ③		
)	②	①	⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③		
+	② +	①	⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③		
(		④	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	
(		⑤	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	④
d	d	⑤	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	④
∩	$d\cap$	⑤	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	④
e	$d\cap e$	⑤	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	④
)	⑤	④	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	$d\cap e\to$ ⑤
÷	⑤ ÷	④	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	$d\cap e\to$ ⑤
f	⑤ ÷ f	④	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	①	$d\cap e\to$ ⑤
)	④	①	② + ④, ⓪	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	⑤ ÷ $f\to$ ④	$d\cap e\to$ ⑤
)	①	⓪	② + ④ $\to$ ①	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	⑤ ÷ $f\to$ ④	$d\cap e\to$ ⑤
→	① →	⓪	② + ④ $\to$ ①	$a:$ ③ $\to$ ②	$b\cdot c\to$ ③	⑤ ÷ $f\to$ ④	$d\cap e\to$ ⑤

out, in reverse preorder:

$$d \cap e \to ⑤$$
$$⑤ \dot{-} f \to ④$$
$$b \cdot c \to ③$$
$$a : ③ \to ②$$
$$② + ④ \to ①$$

and the following symbol 'g' would evoke the final instruction '$① \to g$.'

Böhm's compiler assumed that the source code input would be transliterated into numeric form, but in an Italian patent filed in 1952 he proposed that it should actually be punched on tape using a typewriter with the keyboard shown in Figure 2 [BO 52′, Figure 9]. (His operation '$a \div b$' stood for $|a - b| = (a \dot{-} b) + (b \dot{-} a)$.) Constants in the source program were to be assigned a variable name and input separately. Although Böhm's programs were typeset with square brackets and curly braces as well as parentheses, he noted that only one kind of parenthesis was actually needed.

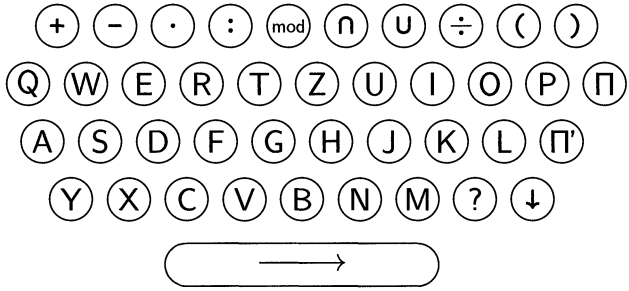


FIGURE 2. Keyboard for program entry, proposed by Corrado Böhm in 1952.

Of all the authors we shall consider, Böhm was the only one who gave an argument that his language was *universal*, namely, capable of computing any computable function.

Meanwhile, in England

Our story so far has introduced us to many firsts: the first algebraic interpreter, the first algorithms for parsing and code generation, the first compiler in its own language, etc. Now we come to the first *real* compiler, in the sense that it was really implemented and used; it really took algebraic statements and translated them into machine language.

The unsung hero of this development was Alick E. Glennie of Fort Halstead, the Royal Armaments Research Establishment. We may justly say "unsung" because it is very difficult to deduce the identity of the person responsible for this pioneering work from the published literature. When Christopher Strachey referred favorably to it in [ST 52, pages 46–47], he did not mention Glennie's name, and it was inappropriate for Glennie to single out his own contributions when he coauthored an article with J. M. Bennett at the time [BG 53, pages 112–113]. In fact, there are apparently only two published references to Glennie's authorship of this early compiler; one of these was a somewhat cryptic remark inserted by an anonymous referee into a review of Böhm's paper [TA 56] while the other appeared in a comparatively inaccessible publication [MG 53].

Glennie called his system AUTOCODE; and it may well have helped to inspire many other "AUTOCODE" routines, of increasing sophistication, developed during the late 1950s. Strachey said that AUTOCODE was beginning to come into use in September 1952. The Manchester Mark I machine language was particularly abstruse — see [WO 51] for an introduction to its complexities, including the intricacies of teleprinter code (used for base-32 arithmetic, backwards) — and its opaqueness may have been why this particular computer witnessed the world's first compiler. Glennie stated his motivations as follows, at the beginning of a lecture that he delivered at Cambridge University in February 1953 [GL 52]:

> The difficulty of programming has become the main difficulty in the use of machines. Aiken has expressed the opinion that the solution of this difficulty may be sought by building a coding machine, and indeed he has constructed one. However it has been remarked that there is no need to build a special machine for coding, since the computer itself being general purpose should be used. ... *To make it easy, one must make coding comprehensible.* This may be done only by improving the notation of programming. Present notations have many disadvantages, all are incomprehensible to the novice, they are all different (one for each machine) and they are never easy to read. It is quite difficult to decipher coded programmes even with notes and even if you yourself made the programme several months ago.
>
> Assuming that the difficulties may be overcome, it is obvious that the best notation for programmes is the usual mathematical notation because it is already known. ... Using a familiar notation for programming has very great advantages, in the elimination of errors in programmes, and the simplicity it brings.

His reference to Aiken should be clarified here, especially because Glennie stated several years later [GL 65] that "I got the concept from a reported idea of Professor Aiken of Harvard, who proposed that a machine be built to make code for the Harvard relay machines." Aiken's coding machine for the Harvard Mark III was cited also by Böhm [BO 52, page 176]; it is described in [HA 52, pages 36–38 and 229–263, illustrated on pages 20, 37, 230]. By pushing appropriate buttons on the console of this proposed device, one or more appropriate machine codes would be punched on tape for the equivalent of three-address instructions such as '$-b3 \times |ci| \to ai$' or '$1/\sqrt{x9} \to r0$'; there was a column of keys for selecting the first operand's sign, its letter name, and its (single) subscript digit, then another column of keys for selecting the function name, etc. (Incidentally, Heinz Rutishauser is listed as one of the 56 authors of the Harvard report [HA 52]; his visit to America in 1950 is one of the reasons he and Böhm did not work jointly together.)

Our TPK algorithm can be expressed in Glennie's AUTOCODE in the following way:

```
01    c@VA t@IC x@½C y@RC z@NC
02    INTEGERS +5→c
03    →t
04    +t TESTA Z
05    -t
06    ENTRY Z
07    SUBROUTINE 6 →z
08    +tt →y →x
09    +tx →y →x
10    +z +cx   CLOSE WRITE 1

11    a@/½ b@MA c@GA d@OA e@PA f@HA i@VE x@ME
12    INTEGERS +20→b +10→c +400→d +999→e +1→f
13    LOOP 10n
14    n →x
15    +b -x →x
16    x →q
17    SUBROUTINE 5 →aq
18    REPEAT n
19    +c →i
20    LOOP 10n
21    +an SUBROUTINE 1 →y
22    +d -y TESTA Z
23    +i SUBROUTINE 3
```

```
24    +e SUBROUTINE 4
25    CONTROL X
26    ENTRY Z
27    +i SUBROUTINE 3
28    +y SUBROUTINE 4
29    ENTRY X
30    +i -f →i
31    REPEAT n
32    ENTRY A CONTROL A WRITE 2 START 2
```

Although this language was much simpler than the Mark I machine code, it was still very machine oriented, as we shall see. (Rutishauser and Böhm had had a considerable advantage over Glennie in that they had designed their own machine language from scratch.) Lines *01–10* of this program represent a subroutine for calculating $f(t)$; '`CLOSE WRITE 1`' on line *10* says that the preceding lines constitute subroutine number 1. The remaining lines yield the main program; '`WRITE 2 START 2`' on line *32* says that the preceding lines constitute subroutine number 2, and that execution starts with number 2.

Let's begin at the beginning of this program and try to give a play-by-play account of what it means. Line *01* is a storage assignment for variables c, t, x, y, and z, in terms of absolute machine locations represented in the beloved teleprinter code. Line *02* assigns the value 5 to c; like all early compiler writers, Glennie shied away from including constants in formulas. Actually his language has been extended here: He had only the statement '`FRACTIONS`' for producing constants between $-\frac{1}{2}$ and $\frac{1}{2}$, assuming that a certain radix point convention was being used on the Manchester machine. Since scaling operations were so complicated on that computer, it would be inappropriate for our purposes to let such considerations mess up or distort the TPK algorithm; thus the `INTEGERS` statement (which is quite in keeping with the spirit of his language) has been introduced to simplify our exposition.

Upon entry to subroutine 1, the subroutine's argument was in the machine's lower accumulator; line *03* assigns it to variable t. Line *04* means "go to label Z if t is positive"; line *05* puts $-t$ in the accumulator; and line *06* defines label Z. Thus the net effect of lines *04–06* is to put $|t|$ into the lower accumulator. Line *07* applies subroutine 6 (integer square root) to this value, and stores it in z. On line *08* we compute the product of t by itself; this fills both upper and lower accumulators, and the upper half (assumed zero) is stored in y, the lower half in x. Line *09* is similar; now x contains t^3. Finally line *10* completes the

calculation of $f(t)$ by leaving $z + 5x$ in the accumulator. The 'CLOSE' operator causes the compiler to forget the meaning of label Z, but the machine addresses of variables c, x, y, and z remain in force.

Line *11* introduces new storage assignments, and in particular it reassigns the addresses of c and x. New constant values are defined on line *12*. Lines *13–18* constitute the input loop, enclosed by LOOP $10n$ ... REPEAT n; here n denotes one of the index registers (the famous Manchester B-lines), the letters k, l, n, o, q, r being reserved for this purpose. Loops in Glennie's language were always done for *decreasing* values of the index, down to and including 0; and in our case the loop was performed for $n = 20, 18, 16, \ldots, 2, 0$. These values are twice what might be expected, because the Mark I addresses were for half-words. Lines *14–16* set index q equal to $20 - n$; this needs to be done in stages (first moving from n to a normal variable, then doing the arithmetic, and finally moving the result to the index variable). The compiler recognized conversions between index variables and normal variables by insisting that all other algebraic statements begin with a + or − sign. Line *17* says to store the result of subroutine 5 (an integer input subroutine) into variable a_q.

Lines *20–31* constitute the output loop. Again n has the value $2i$, so the true value of i has been maintained in parallel with n (see lines *19* and *30*). Line *21* applies subroutine 1, namely our subroutine for calculating $f(t)$, to a_n and stores the result in y. Line *22* branches to label Z if $400 \geq y$; line *25* is an unconditional jump to label X. Line *23* outputs the integer i using subroutine 3, and subroutine 4 on line *24* is assumed to be similar except that a carriage return and line feed are also output. Thus the output is correctly performed by lines *22–29*.

The operations 'ENTRY A CONTROL A' on line *32* define an infinite loop 'A: **go to** A'; this was the so-called *dynamic stop* used to terminate a computation in those good old days.

Our analysis of the sample program is now complete. Glennie's language was an important step forward, but of course it still remained very close to the machine itself. And it was intended for the use of experienced programmers. As he said at the beginning of the user's manual [GL 52′], "The left hand side of the equation represents the passage of information to the accumulator through the adder, subtractor, or multiplier, while the right hand side represents a transfer of the accumulated result to the store." The existence of two accumulators complicated matters; for example, after the multiplication in lines *08* and *09* the upper accumulator was considered relevant (in the notation $\rightarrow y$), while elsewhere only the lower accumulator was used. The expression '$+a+bc$'

meant "load the *lower* accumulator with a, then add it to the double length product bc," whereas '$+bc + a$' meant "form the double-length product bc, then add a into the *upper* half of the accumulator." Expressions like $+ab + cd + ef$ were allowed, but not products of three or more quantities; and there was no provision for parentheses. The language was designed to be used with the 32-character teleprinter code, where there was no distinction between uppercase and lowercase letters. The symbols '+', '−', and '→' were punched as the teleprinter codes 'p', 'm', and '"', respectively.

We have remarked that Glennie's papers have never been published; this may be due to the fact that his employers in the British atomic weapons project were in the habit of keeping documents classified. Glennie's work was, however, full of choice quotes, so it is interesting to repeat several more remarks that he made at the time [GL 52]:

> There are certain other rules for punching that are merely a matter of common sense such as not leaving spaces in the middle of words or misspelling them. I have arranged that such accidents will cause the input programme to exhibit symptoms of distress. ... This consists of the programme coming to a stop and the machine making no further moves.
>
> [The programme] is quite long but not excessively long, about 750 orders. ... The part that deals with the translation of the algebraic notation is the most intricate programme that I have ever devised ... [but the number of orders required] is a small fraction of the total, about 140.
>
> My experience of the use of this method of programming has been rather limited so far, but I have been much impressed by the speed at which it is possible to make up programmes and the certainty of gaining correct programmes. ... The most important feature, I think, is the ease with which it is possible to read back and mentally check the programme. And of course on such features as these will the usefulness of this type of programming be judged.

At the beginning of the user's manual [GL 52′], he mentioned that "the loss of efficiency (in the sense of the additional space taken by routines made with AUTOCODE) is no more than about 10%." This remark appeared also in [BG 53, page 113], and it may well be the source of the oft-heard opinion that compilers are "90% efficient."

On the other hand, Glennie's compiler actually had very little tangible impact on other users of the Manchester machine. For this reason,

Brooker did not even mention it in his 1958 paper entitled "The Autocode programs developed for the Manchester University computers" [BR 58]. This lack of influence may be due in part to the fact that Glennie was not resident at Manchester, but the primary reason was probably that his system did little to solve the really severe problems that programmers had to face in those days of small and unreliable machines. An improvement in the coding process was not regarded then as a breakthrough of any importance, since coding was often the simplest part of a programmer's task. When one had to wrestle with problems of numerical analysis, scaling, and two-level storage, meanwhile adapting one's program to the machine's current state of malfunction, coding itself was quite insignificant.

Thus when Glennie mentioned his system in the discussion following [MG 53], it met with a very cool reception. For example, Stanley Gill's comment reflected the prevailing mood [MG 53, page 79]:

> It seems advisable to concentrate less on the ability to write, say
>
> $$+a+b+ab \rightarrow c$$
>
> as it is relatively easy for the programmer to write
>
> A a
> A b
> H a
> V b
> T c .

Nowadays we would say that Gill had missed a vital point, but in 1953 his remark was perfectly valid.

Some 13 years later, Glennie had the following reflections [GL 65]:

> [The compiler] was a successful but premature experiment. Two things I believe were wrong: (a) Floating-point hardware had not appeared. This meant that most of a programmer's effort was in scaling his calculation, not in coding. (b) The climate of thought was not right. Machines were too slow and too small. It was a programmer's delight to squeeze problems into the smallest space. ...
>
> I recall that automatic coding as a concept was not a novel concept in the early fifties. Most knowledgeable programmers knew of it, I think. It was a well known possibility, like the

possibility of computers playing chess or checkers. ... [Writing the compiler] was a hobby that I undertook in addition to my employers' business: they learned about it afterwards. The compiler ... took about three months of spare time activity to complete.

Early American "Compilers"

None of the authors we have mentioned so far actually used the word "compiler" in connection with what they were doing; the terms were *automatic coding*, *codification automatique*, *Rechenplanfertigung*. In fact it is not especially obvious to programmers today why a compiler should be so called. We can understand this best by considering briefly the other types of programming aids that were in use during those early days.

The first important programming tools to be developed were, of course, general-purpose subroutines for such commonly needed processes as input–output conversions, floating-point arithmetic, and transcendental functions. Once a library of such subroutines had been constructed, people began to think of further ways to simplify programming, and two principal ideas emerged: (a) Coding in machine language could be made less rigid, by using blocks of relocatable addresses [WH 50]. This idea was extended by M. V. Wilkes to the notion of an "assembly routine," able to combine a number of subroutines and to allocate storage [WW 51, pages 27–32]; and Wilkes later [WI 52, WI 53] extended the concept further to include general symbolic addresses that weren't simply relative to a small number of origins. For many years such symbols were called "floating addresses." Similar developments in assembly systems occurred in America and elsewhere (see [RO 52]). (b) An artificial machine language or *pseudocode* was devised, usually providing easy facilities for floating-point arithmetic as if it had been built into the hardware. An "interpretive routine" (sometimes called "interpretative" in those days) would process those instructions, emulating the hypothetical computer. The first interpretive routines appeared in the first textbook about programming, by Wilkes, Wheeler, and Gill [WW 51, pages 34–37, 74–77, 162–164]; the primary aim of their book was to present a library of subroutines and the methodology of subroutine usage. Shortly afterward a refined interpretive routine for floating-point calculation was described by Brooker and Wheeler [BW 53], including a stack by which subroutines could be nested to any depth. Interpretive routines in their more familiar compact form were introduced by J. M. Bennett (see [WW 51, Preface and pages 162–164], [BP 52]); the

most influential was perhaps John Backus's IBM 701 Speedcoding System [BA 54, BH 54]. As we have already remarked, Short Code was a different sort of interpretive routine. The early history of library subroutines, assembly routines, and interpretive routines remains to be written; we have just reviewed it briefly here in order to put the programming language developments into context.

During the latter part of 1951, Grace Murray Hopper developed the idea that pseudocodes need not be interpreted; pseudocodes could also be expanded out into direct machine language instructions. She and her associates at UNIVAC proceeded to construct an experimental program that would do such a translation, and they called it a *compiling routine* [MO 54, page 15].

> To compile means to compose out of materials from other documents. Therefore, the compiler method of automatic programming consists of assembling and organizing a program from programs or routines or in general from sequences of computer code which have been made up previously.

(See also [HO 55, page 22].) The first "compiler" in this sense, named A-0, was in operation in the spring of 1952, when Hopper spoke on the subject at an early ACM National Meeting [HO 52]. Incidentally, M. V. Wilkes came up with a very similar idea, and called it the method of "synthetic orders" [WI 52]; we would now call this a macroexpansion.

The A-0 "compiler" was improved to A-1 (January 1953) and then to A-2 (August 1953); the original implementors were Richard K. Ridgway and Margaret H. Harper. Quite a few references to A-2 appeared in the literature of those days [HM 53, HO 53, HO 53′, MO 54, WA 54], but the authors of those papers gave no examples of the language itself. Therefore it will be helpful to discuss here the state of A-2 as it existed late in 1953, when it was first released to UNIVAC customers for testing [RR 53]. As we shall see, the language was quite primitive by comparison with the ones we have been studying; hence we choose to credit Glennie with the first compiler, although A-0 was completed first. The main point, however, is to understand what was called a "compiler" in 1954, so that we can best appreciate the historical development of programming languages.

Here is how TPK would have looked in A-2 at the end of 1953.

Use of working storage:

00	02	04	06	08	10	12	14..34	36	38	40	42..58
10	5	400	−1	∞	4	3	$a_0..a_{10}$	i	y, y', y''	t, t', t''	buffer

Program:

```
0.  GMI000 000002
    ITEM01 WS.000   Read input and necessary constants from tape 2
    SERVQ2 BLQCKA
    1RG000 000000
1.  GMM000 000001
    000180 020216   10.0 = i
    1RG000 001000
2.  AM0034 034040   a_10^2 = t
3.  RNA040 010040   ∜t = t'
4.  APN034 012038   a_10^3 = y
5.  AM0002 038038   5y = y'
6.  AA0040 038038   t' + y' = y''
7.  AS0004 038040   400 − y'' = t''
8.  QWNΔCQ DEΔ003
    K00000 K00000
    F00912 E001RG   if t'' ≥ 0, go on to Op. 10
    000000 Q001CN
    1RG000 008040
    1CN000 000010
9.  GMM000 000001
    000188 020238   'ΔΔΔTQQ ΔLARGE ΔΔΔΔΔΔ ΔΔΔΔΔΔ' = y''
    1RG000 009000
10. YTQ036 038000   Print i, y''
11. GMM000 000001
    000194 200220   Move 20 words from WS14 to WS40
    1RG000 011000
12. GMM000 000001
    000220 200196   Move 20 words from WS40 to WS16
    1RG000 012000
13. ALL012 F000T⌿
    1RG000 013036   Replace i by i + (−1)
    2RG000 000037     and go to Op. 2 if i > −1,
    3RG000 000006     otherwise go to Op. 14
    4RG000 000007
    5RG000 000006
    6RG000 000007
    1CN000 000002
    2CN000 000014
    1RS000 000036
    2RS000 000037
14. QWNΔCQ DEΔ002
    810000 820000   Rewind tapes 1 and 2, and halt.
    900000 900000
    1RG000 014000
    ⌿ØENDΔ INFQ.⌿
```

There were 60 words of working storage, and each floating-point number used two words. These working storages were usually addressed by numbers 00, 02, ..., 58, except in the GMM instruction (move generator) when they were addressed by 180, 182, ..., 238, respectively; see operations 1, 9, 11, and 12. Since there was no provision for absolute value, operations 2 and 3 of this program find $\sqrt{|a_{10}|}$ by computing $\sqrt[4]{a_{10}^2}$. (The A-2 compiler would replace most operators by a fully expanded subroutine, in line; this subroutine would be copied anew each time it was requested, unless it was one of the four basic floating-point arithmetic operations.) Since there was no provision for subscripted variables, operations 11 and 12 shift the array elements after each iteration.

Most arithmetic instructions were specified with a three-address code, as shown in operations 2–7. But at this point in the development of A-2 there was no way to test the relation '$\geq$' without resorting to machine language; only a test for equality was built in. So operation 8 specifies the necessary UNIVAC instructions. (The first word in operation 8 says that the following 003 lines contain UNIVAC code. Those three lines extract (E) the sign of the first numeric argument (1RG) using a system constant in location 912, and if it was positive they instruct the machine to go to program operator 1CN. The next two lines say that 1RG is to be t'' (working storage 40), and that 1CN is to be the address of operation 10. The '008' in the 1RG specification tells the compiler that this is operation 8; such redundant information was checked at compile time. The compiler would substitute appropriate addresses for 1RG and 1CN in the machine language instructions. Since there was no notation for '1RG+1', the programmer had to supply ten different parameter lines to the increment-and-test routine in operation 13.)

By 1955 A-2 had become more streamlined, and the necessity for 'QWN CQDE' in TPK had disappeared; operations 7–14 could now become

```
7.  QT0038 004000   To Op. 9 if y'' > 400
    1CN000 000009
8.  QU0038 038000   Go to Op. 10
    1CN000 000010
9.  MV0008 001038 ┐
10. YTQ036 038000 │
11. MV0014 010040 │
12. MV0040 010016 │
13. AAL036 006006 } Same meaning as before, but new syntax.
    1CN000 000002 │
    2CN000 000014 │
14. RWS120 000000 │
    ENDACQ DINGAA ┘
```

(A description of A-2 coding, vintage 1955, appears in [PR 55], and also in [TH 55], which presented the same example program.)

Laning and Zierler

Grace Hopper was particularly active as a spokesperson for automatic programming during the 1950s; she went barnstorming throughout the country, significantly helping to accelerate the rate of progress. One of the most important things she accomplished was to help organize two key symposia on the topic, in 1954 and 1956, under the sponsorship of the Office of Naval Research. These symposia brought together many people and ideas at an important time. (On the other hand, it must be remarked that the contributions of Zuse, Curry, Burks, Mauchly, Böhm, and Glennie were not mentioned at either symposium, and Rutishauser's work was cited only once — not quite accurately [GO 54, page 76]. Communication was not rampant!)

In retrospect, the biggest event of the 1954 symposium on automatic programming was the announcement of a system that J. Halcombe Laning, Jr., and Niel Zierler had recently implemented for the Whirlwind computer at MIT. However, the significance of that announcement is not especially evident from the published proceedings [NA 54], 97% of which are devoted to enthusiastic descriptions of assemblers, interpreters, and 1954-style "compilers." We know of the impact mainly from Grace Hopper's introductory remarks at the 1956 symposium, discussing the past two years of progress [HO 56]:

> A description of Laning and Zierler's system of algebraic pseudocoding for the Whirlwind computer led to the development of Boeing's BACAIC for the 701, FORTRAN for the 704, AT-3 for the UNIVAC, and the Purdue System for the Datatron and indicated the need for far more effort in the area of algebraic translators.

A clue to the importance of Laning and Zierler's contribution can also be found in the closing pages of a paper by John Backus and Harlan Herrick at the 1954 symposium. After describing IBM 701 Speedcoding and the tradeoffs between interpreters and "compilers," they concluded by speculating about the future of automatic programming [BH 54]:

> A programmer might not be considered too unreasonable if he were willing only to produce the formulas for the numerical solution of his problem, and perhaps a plan showing how the data was to be moved from one storage hierarchy to another, and then

> demand that the machine produce the results for his problem. No doubt if he were too insistent next week about this sort of thing he would be subject to psychiatric observation. However, next year he might be taken more seriously.

After listing numerous advantages of high-level languages, they said: "Whether such an elaborate automatic-programming system is possible or feasible has yet to be determined." As we shall soon see, the system of Laning and Zierler proved that such a system is indeed possible.

Brief mention of their system was made by Charles Adams at the symposium [AL 54]; but the full user's manual [LZ 54] ought to be reprinted someday because their language went so far beyond what had been implemented before. The programmer no longer needed to know much about the computer at all, and the user's manual was (for the first time) addressed to a complete novice. Here is how TPK would look in their system:

```
01        v|N = ⟨input⟩,
02        i = 0,
03    1   j = i + 1,
04        a|i = v|j,
05        i = j,
06        e = i − 10.5,
07        CP 1,
08        i = 10,
09    2   y = F¹(F¹¹(a|i)) + 5(a|i)³,
10        e = y − 400,
11        CP 3,
12        z = 999,
13        PRINT i, z.
14        SP 4,
15    3   PRINT i, y.
16    4   i = i − 1,
17        e = −0.5 − i,
18        CP 2,
19        STOP
```

The program was typed on a Flexowriter, a machine that punched paper tape and had a fairly large character set (including both uppercase and lowercase letters); at MIT they also had superscript digits ^{0}, ^{1}, ..., ^{9} and a vertical line |. The language used the vertical line to indicate *subscripts*; thus the '$5(a|i)^3$' on line *09* means $5a_i^3$.

A programmer would insert eleven input values for the TPK algorithm into the place shown on line *01*; those values would be converted to binary notation and stored on the magnetic drum as variables v_1, v_2, ..., v_{11}. If the numbers had a simple arithmetic pattern, an abbreviation could also be used; for example,

$$v|N = 1\,(.5)\,2\,(.25)\,3.5\,(1)\,5.5$$

would set $(v_1, \ldots, v_{11}) \leftarrow (1, 1.5, 2, 2.25, 2.5, 2.75, 3, 3.25, 3.5, 4.5, 5.5)$. If desired, a special code could be punched on the Flexowriter tape in line *01*, allowing the operator to substitute a data tape at that point before reading in the rest of the source program.

Lines *02–07* are a loop that moves the variables $v_1, \ldots, v_{11}$ from the drum to variables $a_0, \ldots, a_{10}$ in core. (All variables were in core unless specifically assigned to the drum by an ASSIGN or $|N$ instruction. This was an advanced feature of the system not needed in small problems.) The only thing that isn't self-explanatory about lines *02–07* is line *07*; 'CP k' means, "if the last expression computed was negative, go to the instruction labeled k."

In line *09*, F^1 denotes square root and F^{11} denotes absolute value. In line *14*, 'SP' denotes an unconditional jump. (CP and SP were the standard mnemonics for jumps in Whirlwind machine language.) Thus, except for control statements — for which there was no existing mathematical convention — Laning and Zierler's notation was quite easy to read. Their expressions featured normal operator precedence, as well as implied multiplication and exponentiation; and they even included a built-in Runge–Kutta mechanism for integrating a system of differential equations if the programmer wrote formulas such as

$$\mathrm{D}x = y + 1,$$
$$\mathrm{D}y = -x,$$

where D stands for d/dt! Another innovation, designed to help debugging, was to execute statement number 100 after any arithmetic error message, if 100 was a PRINT statement.

According to [LM 70], Laning first wrote a prototype algebraic translator in the summer of 1952. He and Zierler had extended it to a usable system by May 1953, when the Whirlwind had only 1024 16-bit words of core memory in addition to its drum. The version described in [LZ 54] utilized 2048 words and drum, but earlier compromises due to such extreme core limitations caused it to be quite slow. The source code was

translated into blocks of subroutine calls, stored on the drum, and after being transferred to core storage (one equation's worth at a time) these subroutines invoked the standard floating-point interpretive routines on the Whirlwind [AL 54, page 64].

> The use of a small number of standard closed subroutines has certain advantages of logical simplicity; however, it also often results in the execution of numerous unnecessary operations. This fact, plus the frequent reference to the drum required in calling in equations, results in a reduction of computing speed of the order of magnitude of ten to one from an efficient computer program.

From a practical standpoint, those were damning words. Laning recalled, eleven years later [LA 65], that

> This was in the days when machine time was king, and people-time was worthless (particularly since I was not even on the Whirlwind staff). ... [The program] did perhaps pay for itself a few times when a complex problem required solutions with a twenty-four hour deadline.

In a recent search of his files, Laning found a listing of the Whirlwind compiler's first substantial application [LA 76]:

> The problem addressed is that of a three-dimensional lead pursuit course flown by one aircraft attacking another, including the fire control equations. What makes this personally interesting to me is tied in with the fact that for roughly five years previous to this time the [MIT Instrumentation] Lab had managed and operated the MIT Rockefeller Differential Analyzer with the principal purpose of solving this general class of problem. Unfortunately, the full three dimensional problem required more integrators than the RDA possessed.
>
> My colleagues who formulated the problem were very skeptical that it could be solved in any reasonable fashion. As a challenge, Zierler and I sat down with them in a $2\frac{1}{2}$ hour coding session, at least half of which was spent in defining notation. The tape was punched, and with the usual beginner's luck it ran successfully the first time! Although we never seriously capitalized on this capability, for reasons of cost and computer availability, my own ego probably never before or since received such a boost.

The lead-pursuit source program consisted of 79 statements, including 29 that merely assigned initial data values. The remaining statements included seven uses of the differential equation feature.

Laning describes his original parsing technique as follows [LA 76]:

> Nested parentheses were handled by a sequence of generated branch instructions (SP). In a one-pass operation the symbols were read and code generated a symbol at a time; the actual execution sequence used in-line SP orders to hop about from one point to another. The code used some rudimentary stacks, but was sufficiently intricate that I didn't understand it without extreme concentration even when I wrote it. ... Structured programs were not known in 1953!
>
> The notion of operator precedence as a formal concept did not occur to me at the time; I lived in fear that someone would write a perfectly reasonable algebraic expression that my system would not analyze correctly.

Plans for a much expanded Whirlwind compiler were dropped when the MIT Instrumentation Lab acquired its own computer, an IBM 650. Laning and his colleagues Philip C. Hankins and Charles P. Werner developed a compiler called MAC for this machine in 1957 and 1958. Although MAC falls out of the time period covered by our story, it deserves brief mention here because of its unusual three-line format proposed by R. H. Battin circa 1956, somewhat like Zuse's original language. For example, the statement

```
E |                                3
M | Y = SQRT(ABS(A   )) + 5 A
S |                  I+1         I+1
```

would be punched on three cards. Although this language has not become widely known, it was very successful locally: MAC compilers were ultimately developed for use with IBM 704, 709, 7090, and 360 computers, as well as the Honeywell H800 and H1800 and the CDC 3600. (See [LM 70].) "At the present time, MAC and FORTRAN have about equal use at CSDL," according to [LA 76], written in 1976; here CSDL means C. S. Draper Laboratory, the successor to MIT's Instrumentation Lab.

But we had better get back to our story of the early days.

FORTRAN 0

During the first part of 1954, John Backus began to assemble a group of people within IBM Corporation to work on improved systems of

automatic programming (see [BA 76]). Shortly after learning of the Laning and Zierler system at the ONR meeting in May, Backus wrote to Laning that "our formulation of the problem is very similar to yours: however, we have done no programming or even detailed planning." Within two weeks, Backus and his co-workers Harlan Herrick and Irving Ziller visited MIT in order to see the Laning–Zierler system in operation. The big problem facing them was to implement such a language with suitable efficiency [BH 64, page 382]:

> At that time, most programmers wrote symbolic machine instructions exclusively (some even used absolute octal or decimal machine instructions). Almost to a man, they firmly believed that any mechanical coding method would fail to apply that versatile ingenuity which each programmer felt he possessed and constantly needed in his work. Therefore, it was agreed, compilers could only turn out code which would be intolerably less efficient than human coding (intolerable, that is, unless that inefficiency could be buried under larger, but desirable, inefficiencies such as the programmed floating-point arithmetic usually required then). ...
>
> [Our development group] had one primary fear. After working long and hard to produce a good translator program, an important application might promptly turn up which would confirm the views of the skeptics: ... its object program would run at half the speed of a hand-coded version. It was felt that such an occurrence, or several of them, would almost completely block acceptance of the system.

By November 1954, Backus's group had specified "The IBM Mathematical FORmula TRANslating system, FORTRAN." (Almost all the languages we shall discuss from now on had acronyms.) The first paragraph of their report [IB 54] emphasized that previous systems had offered the choice of easy coding and slow execution or laborious coding and fast execution, but FORTRAN would provide the best of both worlds. It also placed specific emphasis on the IBM 704; machine independence was not a primary goal, although a concise mathematical notation that "does not resemble a machine language" was definitely considered important. Furthermore they stated that "each future IBM calculator should have a system similar to FORTRAN accompanying it" [IB 54].

> It is felt that FORTRAN offers as convenient a language for stating problems for machine solution as is now known. ...

> After an hour course in FORTRAN notation, the average programmer can fully understand the steps of a procedure stated in FORTRAN language without any additional comments.

They went on to describe the considerable economic advantages of programming in such a language.

Readers of the present account probably imagine that they know FORTRAN already, because FORTRAN is certainly the earliest high-level language that is still in use. But comparatively few people have seen the original 1954 version of the language, so it is instructive to study TPK as it might have been expressed in "FORTRAN 0":

```
01              DIMENSION A(11)
02              READ A
03          2   DO 3,8,30 J=1,11
04          3   I = 11-J
05              Y=SQRT(ABS(A(I+1)))+5*A(I+1)**3
06              IF (400 >= Y) 8,4
07          4   PRINT I,999.
08              GO TO 2
09          8   PRINT I,Y
10         30   STOP
```

The `READ` and `PRINT` statements here do not mention any formats, although an extension to format specification was contemplated [page 26]; programmer-defined functions were also under consideration [page 27]. The `DO` statement in line *03* means, "Do statements 3 thru 8 and then go to statement 30"; the abbreviation '`DO 8 J =1,11`' was also allowed at that time, but the original general form is shown here for fun. Notice that the `IF` statement was originally only a two-way branch (line *06*); the relation could be `=`, `>`, or `>=`. On line *05* we note that function names need not end in `F`; they were required to be at least three characters long, and there was no maximum limit (except that expressions could not be longer than 750 characters). Conversely, the names of variables were restricted to be at most *two* characters long at this time; but this in itself was an innovation: FORTRAN was the first algebraic language in which a variable's name could be longer than one letter, contrary to established mathematical conventions. Note also that mixed mode arithmetic was allowed; the compiler would convert '5' to '5.0' in line *05*. A final curiosity about this program is the `GO TO` on line *08*; this statement did not begin the `DO` loop all over again, it merely initiated the next iteration.

Several things besides mixed mode arithmetic were allowed in FORTRAN 0 but withdrawn during implementation, notably: (a) one level

of subscripted subscripts, such as `A(M(I,J),N(K,L))` were allowed; (b) subscripts of the form `N*I+J` were permitted, provided that at least two of the variables N, I, J were declared to be "relatively constant" (that is, infrequently changing); (c) a `RELABEL` statement was intended to permute array indices cyclically without physically moving the array in storage. For example, '`RELABEL A(3)`' was to be like setting $(A_1, A_2, A_3, \ldots, A_n) \leftarrow (A_3, \ldots, A_n, A_1, A_2)$.

Incidentally, statements were called *formulas* throughout the 1954 document. There were arithmetic formulas, `DO` formulas, `GO TO` formulas, etc. Similar terminology had been used by Böhm, while Laning and Zierler and Glennie spoke of "equations"; Grace Hopper called them "operations." Furthermore, the word "compiler" is never used in [IB 54]; there is a FORTRAN language and a FORTRAN system, but not a FORTRAN compiler.

The FORTRAN 0 document represents the first attempt to define the syntax of a programming language rigorously. Backus's important notation [BA 59], which eventually became 'BNF' [KN 64], can be seen in embryonic form here.

With the FORTRAN language defined, it "only" remained to implement the system. It is clear from reading [IB 54] that considerable plans had already been made toward the implementation during 1954; but the full job took 2.5 more years (18 person-years) [BA 79], so we shall leave the IBM group at work while we consider other developments.

Brooker's AUTOCODE

Back in Manchester, R. A. Brooker introduced a new type of AUTOCODE for the MARK I machine. This language was much "cleaner" than Glennie's, being nearly machine-independent and using programmed floating-point arithmetic. But it allowed only one operation per line, there were few mnemonic names, and there was no way for a user to define subroutines. The first plans for this language, as of March 1954, appeared in [BR 55], and the language eventually implemented was almost the same [BR 56, pages 155–157]. Brooker's emphasis on economy of description when defining the rules of AUTOCODE was especially noteworthy: "What the author aimed at was two sides of a foolscap sheet with possibly a third side to describe an example" [BR 55].

The floating-point variables in Brooker's language were called $v1$, $v2$, ..., and the integer variables — which may be used also as indices (subscripts) — were called $n1$, $n2$, ... The AUTOCODE for TPK is easily readable with only a few auxiliary comments, given the memory

assignments $a_i \equiv v_{1+i}$, $y \equiv v_{12}$, $i \equiv n_2$:

6	$n1$	$= 1$	sets $n_1 = 1$
1	$vn1$	$= \mathrm{I}$	reads input into v_{n_1}
	$n1$	$= n1 + 1$	
	$j1, 11$	$\geq n1$	jumps to 1 if $n_1 \leq 11$
	$n1$	$= 11$	
2	$*n2$	$= n1 - 1$	prints $i = n_1 - 1$
	$v12$	$= vn1$	
	$j3, v12$	$\geq 0{\cdot}0$	
	$v12$	$= 0{\cdot}0 - v12$	sets $v_{12} = \lvert v_{12} \rvert$
3	$v12$	$= \mathrm{F1}(v12)$	$(v_{12} = \sqrt{\lvert a_i \rvert})$
	$v13$	$= 5{\cdot}0 \otimes vn1$	
	$v13$	$= vn1 \otimes v13$	
	$v13$	$= vn1 \otimes v13$	$(v_{13} = 5a_i^3)$
	$v12$	$= v12 + v13$	$(y = f(a_i))$
	$j4, v12$	$> 400{\cdot}0$	
	$*v12$	$= v12$	prints y
	$j5$		
4	$*v12$	$= 999{\cdot}0$	prints 999
5	$n1$	$= n1 - 1$	
	$j2, n1$	> 0	tests for last cycle
	H		halts
	$(j6)$		starts programme

The final instruction illustrates an interesting innovation: An instruction or group of instructions in parentheses was obeyed immediately, rather than added to the program. Thus '$(j6)$' jumps to statement 6.

This language is not at a very high level, but Brooker's main concern was simplicity and a desire to keep information flowing smoothly to and from the electrostatic high-speed memory. Mark I's electrostatic memory consisted of only 512 20-bit words, and it was necessary to make frequent transfers from and to the 32K-word drum; floating-point subroutines could compute while the next block of program was being read in. Thus two of the principal difficulties facing a programmer — scaling and coping with the two-level store — were removed by his AUTOCODE system, and it was heavily used. For example [BR 58, page 16]:

> Since its completion in 1955 the Mark I AUTOCODE has been used extensively for about 12 hours a week as the basis of a computing service for which customers write their own programs and post them to us.

George E. Felton, who developed the first AUTOCODE for the Ferranti Pegasus, says in [FE 60] that its specification "clearly owes much to Mr. R. A. Brooker." Incidentally, Brooker's next AUTOCODE—for the Mark II or "Mercury" computer, first delivered in 1957—was considerably more ambitious; see [BR 58, BR 58′, BR 60].

Russian Programming Programs

Work on automatic programming began in Russia at the Mathematical Institute of the Soviet Academy of Sciences, and at the Academy's computation center, which originally was part of the Institute of Exact Mechanics and Computing Technique. The early Russian systems were appropriately called programming programs [Programmiruîòshchye Programmy]—or ПП for short. An experimental program ПП-1 for the STRELA computer was constructed by E. Z. Lîûbimskiĭ and S. S. Kamynin during the summer of 1954; and these two authors, together with M. R. Shura-Bura, É. S. Lukhovitskaîà, and V. S. Shtarkman, completed a production compiler called ПП-2 in February 1955. Their compiler is described in [KL 58]. Meanwhile, A. P. Ershov began in December 1954 to design another programming program, for the BESM computer, with the help of L. N. Korolev, L. D. Panova, V. D. Podderîûgin, and V. M. Kurochkin; their compiler, called simply ПП, was completed in March 1956, and it is described in Ershov's book [ER 58]. A review of these developments appears in [KO 58].

In both of these cases, and in the later system ПП-С completed in 1957 (see [ER 58′]), the language was based on a notation for expressing programs developed by A. A. Lîàpunov in 1953. Lîàpunov's operator schemata [LJ 58] provide a concise way to represent program structure in a linear manner; in some ways this approach is analogous to the ideas of Curry that we have already considered, but it is somewhat more elegant and it became widely used in Russia.

Let us consider first how the TPK algorithm (exclusive of input-output) can be described in ПП-2. The overall operator scheme for the program would be written

$$A_1 \underset{13}{\lrcorner} Z_2 A_3 R_4 \overset{6}{\ulcorner} A_5 \overset{4}{\urcorner} A_6 R_7 \underset{10}{\llcorner} A_8 N_9 \overset{11}{\ulcorner} \underset{7}{\lrcorner} A_{10} \overset{9}{\urcorner} A_{11} F_{12} R_{13} \underset{2}{\llcorner} N_{14}.$$

Here the operators are numbered 1 through 14; the notations

$$\overset{n}{\ulcorner} \quad \text{and} \quad \underset{m}{\llcorner}$$

mean "go to operator n if true" and "go to operator m if false," respectively, while

$$\overset{i}{\urcorner} \quad \text{and} \quad \underset{i}{\llcorner}$$

are the corresponding notations for "coming from operator i." This operator scheme was not itself input to the programming program explicitly; it would be kept by the programmer as documentation in lieu of a flowchart. The details of operators would be written separately and input to ΠΠ-2 after dividing them into operators of types R (relational), A (arithmetic), Z (dispatch), F (address modification), O (restoration), and N (nonstandard, that is, machine language). In our example the details are essentially this:

R_4. $p_1; 6, 5$	[if p_1 is true go to 6 else to 5]
R_7. $p_2; 8, 10$	[if p_2 is true go to 8 else to 10]
R_{13}. $p_3; 14, 2$	[if p_3 is true go to 14 else to 2]
p_1. $c_3 < v_2$	$[0 < x]$
p_2. $c_4 < v_3$	$[400 < y]$
p_3. $v_6 < c_3$	$[i < 0]$
A_1. $c_6 = v_6$	[$10 = i$, that is, set i equal to 10]
A_3. $v_1 = v_2$	$[a_i = x]$
A_5. $c_3 - v_2 = v_2$	$[0 - x = x]$
A_6. $(\sqrt{v_2}) + (c_5 \cdot v_1 \cdot v_1 \cdot v_1) = v_3$	$[\sqrt{x} + (5 \cdot a_i \cdot a_i \cdot a_i) = y]$
A_8. $v_6 = v_4,\ c_2 = v_5$	$[i = b_{20-2i},\ 999 = b_{21-2i}]$
A_{10}. $v_6 = v_4,\ v_3 = v_5$	$[i = b_{20-2i},\ y = b_{21-2i}]$
A_{11}. $v_6 - c_1 = v_6$	$[i - 1 = i]$
Z_2. $v_1; 3, 6$	[dispatch a_i to special cell, in operators 3 thru 6]
F_{12}. $v_6; 2, 10$	[modify addresses depending on parameter i, in operators 2 thru 10]
N_9. BP 11	[go to operator 11]
N_{14}. OST	[stop]
Dependence on parameter v_6.	$v_1, v_1, -1;\quad v_4, v_5, +2$ [when i changes, v_1 goes down by 1, and v_4 thru v_5 go up by 2]
c_1. $.1 \cdot 10^1$	[1]
c_2. $.999 \cdot 10^3$	[999]
c_3. 0	[0]
c_4. $.4 \cdot 10^3$	[400]
c_5. $.5 \cdot 10^1$	[5]
c_6. $.1 \cdot 10^2$	[10]
Working cells: 100, 119	[the compiled program can use locations 100–119 for temporary storage]

v_1.	130	[initial address of a_i]
v_2.	131	[address of x]
v_3.	132	[address of y]
v_4.	133	[initial address of b_{20-2i}]
v_5.	134	[initial address of b_{21-2i}]
v_6.	154	[address of i]

Operator 1 initializes i, then operators 2–13 are the loop on i. Operator 2 moves a_i to a fixed cell of memory, and makes sure that operators 3–6 use that fixed cell; this programmer-supplied optimization reduces the number of addresses in instructions that have to be modified when i changes. Operators 3–5 set $x = |a_i|$, and operator 6 sets $y = f(a_i)$. (Notice the parentheses in operator 6; precedence was not recognized.) Operators 7–10 store the desired outputs in memory. Operators 11 and 12 decrease i and appropriately adjust the addresses of quantities that depend on i. Operators 13 and 14 control looping and stopping.

The algorithms used in ПП-2 are quite interesting from the standpoint of compiler history. For example, they avoided the recomputation of common subexpressions within a single formula, and they carefully optimized the use of working storage. They also produced efficient code for relational operators compounded from a series of elementary relations, so that, for example,

$$(p_1 \vee (p_2 \cdot p_3) \vee \overline{p_4}) \cdot p_5 \vee p_6$$

would be compiled as shown in Figure 3.

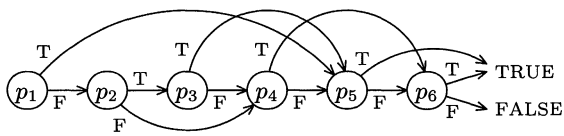


FIGURE 3. Optimization of Boolean expressions in an early Russian compiler.

Ershov's ПП language improved on ПП-2 in several respects, notably: (a) the individual operators did not need to be numbered, and they could be intermixed in the natural sequence; (b) no address modification needed to be specified, and there was a special notation for loops; (c) the storage for variables was allocated semiautomatically; (d) operator precedence could be used to reduce the number of parentheses within expressions. The TPK algorithm looks like this in ПП:

01	Massiv a (11 îàcheek)	[declares an array of 11 cells]
02	$a_0 = 0$	[address in array a]

03 $a_j = -1 \cdot j + 10$ [address in array a depending on j]
04 $j : j_{\text{nach}} = 0,\ j_{\text{kon}} = 11$ [information on loop indexes]
05 $0, 11, 10, 5, y, 400, 999, i$ [the remaining constants and variables]

06 $(\text{Ma}, 080, 0, a_0);\ (\text{Mb}, 0, 0\overline{1}, 0);$

07 $\left[\begin{smallmatrix} \\ j \end{smallmatrix}\right. 10 - j \Rightarrow i;\ \sqrt{\ } \bmod a_j + 5 \times a_j^3 \Rightarrow y;$

08 $\text{R}(y, 0102; \overset{0101}{\lceil} (400, \infty));$

09 $\underset{0101}{\llcorner}$ Vyd $i, \Rightarrow 0;$ Vyd $999, \Rightarrow 0;$ $\overset{0103}{\lceil};$

10 $\underset{0102}{\llcorner}$ Vyd $i, \Rightarrow 0;$ Vyd $y, \Rightarrow 0;$ $\underset{0103}{\llcorner}\ ];$ stop.

After declarations on lines *01–05*, the program appears here on lines *06–10*. In ПП each loop was associated with a different index name, and the linear dependence of array variables on loop indices was specified as in line *03*; the notation a_j does not mean the jth element of a, it means an element of a that *depends* on j. The commands in line *06* are BESM machine language instructions that read 11 words into memory starting at a_0. Line *07* shows the beginning of the loop on j, which ends at the '$]$' on line *10*; all loop indices must step by $+1$. (The initial and final-plus-one values for the j loop are specified on line *04*.) Line *08* is a relational operator that means, "If y is in the interval $(400, \infty)$, that is, if $y > 400$, go to label 0101; otherwise go to 0102." Labels were given as hexadecimal numbers, and the notation $\underset{n}{\llcorner}$ stands for the program location of label n. The 'Vyd' instruction in lines *09* and *10* means convert to decimal, and '$\Rightarrow 0$' means print. Everything else should be self-explanatory.

The Russian computers had no alphabetic input or output, so the programs written in ПП-2 and ПП were converted into numeric codes. This was a rather tedious and intricate process, usually performed by two specialists who would compare their independent hand transliterations in order to prevent errors. As an example of this encoding process, here is how the program above would actually have been converted into BESM words in the form required by ПП. [Hexadecimal digits were written 0, 1, ..., 9, $\overline{0}$, $\overline{1}$, ..., $\overline{5}$. A 39-bit word in BESM could be represented either in instruction format,

$$bbh\ bqhh\ bqhh\ bqhh,$$

where b denotes a binary digit (0 or 1), q a quaternary digit (0, 1, 2, or 3), and h a hexadecimal digit; or in floating-binary numeric format,

$$\pm\, 2^k,\ hh\ hh\ hh\ hh,$$

where k is a decimal number between -32 and $+31$ inclusive. Both of these representations were used at various times in the encoding of a IIII program, as shown below.]

Location	*Contents*	*Meaning*
07	000 0000 0000 0000	no space needed for special subroutines
08	000 0000 0000 0013	last entry in array descriptor table
09	000 0000 0000 0015	first entry for constants and variables
$0\bar{0}$	000 0000 0000 001$\bar{2}$	last entry for constants and variables
$0\bar{1}$	000 0000 0000 002$\bar{5}$	base address for encoded program scheme
$0\bar{2}$	000 0000 0000 0042	last entry of encoded program
$0\bar{3}$	000 0000 0000 029$\bar{5}$	base address for "block γ"
$0\bar{4}$	000 0000 0000 02$\bar{1}\bar{5}$	base address for "block α"
$0\bar{5}$	000 0000 0000 02$\bar{3}\bar{5}$	base address for "block β"
10	01$\bar{5}$ 0000 000$\bar{1}$ 0000	a = array of size 11
11	000 1001 0000 0000	coefficient of -1 for linear dependency
12	2^1, 00 00 00 00	$a_0 = 0$ relative to a
13	2^2, 14 00 00 $0\bar{0}$	$a_j = -1 \cdot j + 10$ relative to a
14	000 0015 0016 0000	j = loop index from 0 to 11
15	2^{-32}, 00 00 00 00	0
16	2^4, $\bar{1}0$ 00 00 00	11
17	2^4, $\bar{0}0$ 00 00 00	10
18	2^3, $\bar{0}0$ 00 00 00	5
19	2^9, $\bar{2}8$ 00 00 00	400
$1\bar{0}$	2^{10}, $\bar{5}9$ $\bar{2}0$ 00 00	999
$1\bar{1}$	000 0000 0000 0000	i
$1\bar{2}$	000 0000 0000 0000	y
30	016 0080 0000 0012	(Ma, 080, 0, a_0)
31	017 0000 000$\bar{1}$ 0000	(Mb, 0, $0\bar{1}$, 0)
32	018 0014 0000 0000	$\lfloor_j$
33	2^0, 17 04 14 08	$10 - j \Rightarrow$
34	2^0, $1\bar{1}$ $\bar{5}3$ $\bar{5}\bar{2}$ 13	$i \surd \bmod a_j$
35	2^0, 03 18 09 13	$+ 5 \times a_j$
36	2^0, $0\bar{2}$ 08 $1\bar{2}$ 00	$^3 \Rightarrow y$
37	018 0000 001$\bar{2}$ 0102	R(y, 0102;
38	008 0019 0000 0101	$\overset{0101}{\lceil}$ (400, ∞))
39	018 0101 0000 0000	$\underset{0101}{\lfloor}$
$3\bar{0}$	2^0, $\bar{5}4$ $1\bar{1}$ 07 00	Vyd i, $\Rightarrow$ 0
$3\bar{1}$	2^0, $\bar{5}4$ $1\bar{0}$ 07 00	Vyd 999, $\Rightarrow$ 0
$3\bar{2}$	01$\bar{1}$ 0000 0000 0103	$\overset{0103}{\lceil}$
$3\bar{3}$	018 0102 0000 0000	$\underset{0102}{\lfloor}$
$3\bar{4}$	2^0, $\bar{5}4$ $1\bar{1}$ 07 00	Vyd i, $\Rightarrow$ 0

```
35̄  2^0, 5̄4 1̄2 07 00      Vyd y, ⇒ 0
40  018 0103 0000 0000    └──┘
                          0103
41  015̄ 135̄5̄ 135̄5̄ 135̄5̄   ]
42  015̄ 0000 0000 0000    stop
```

The BESM had 1024 words of core memory, plus some high-speed read-only memory, and a magnetic drum holding 5×1024 words. The ПП compiler worked in three passes (formulas and relations, loops, final assembly) and it contained a total of 1200 instructions plus 150 constants. Ershov published detailed specifications of its structure and all its algorithms in [ER 58], a book that was extremely influential at the time — for example, a Chinese translation was published in Peking in 1959. The book shows that Ershov was aware of Rutishauser's work [page 9], but he gave no other references to non-Russian sources.

A Western Development

Computer professionals at the Boeing Airplane Company in Seattle, Washington, felt that "in this jet age, it is vital to shorten the time from the definition of a problem to its solution." So they introduced BAC-AIC, the Boeing Airplane Company Algebraic Interpretive Computing system for the IBM 701 computer.

BACAIC was an interesting language and compiler developed by Mandalay Grems and R. E. Porter, who began work on the system in the latter part of 1954; they presented it at the Western Joint Computer Conference held in San Francisco, in February 1956 [GP 56]. Although the 'I' in BACAIC stands for "interpretive," their system actually translated algebraic expressions into machine-language calls on subroutines, with due regard for parentheses and precedence, so we would now call it a compiler.

The BACAIC language was unusual in several respects, especially in its control structure, which assumed one-level iterations over the entire program; a program was considered to be a nearly straight-line computation to be applied to various "cases" of data. There were no subscripted variables. However, the TPK algorithm could be performed by inputting the data in reverse order using the following program:

1. $I - K1 * I$
2. X
3. WHN X GRT $K2$ USE 5
4. $K2 - X * 2$
5. SRT $2 + K3$. X PWR $K4$

6. WHN 5 GRT $K5$ USE 8
7. TRN 9
8. $K6 * 5$
9. TAB I 5

Here '$*$' is used for assignment, and '.' for multiplication; variables are given single-letter names (except K), and constants are denoted by $K1$ through $K99$. The program above is to be used with the following input data:

Case 1. $K1 = 1.0 \quad K2 = 0.0 \quad K3 = 5.0 \quad K4 = 3.0$
$K5 = 400.0 \quad K6 = 999.0 \quad I = 11.0 \quad X = a_{10}$
Case 2. $X = a_9$
Case 3. $X = a_8$
$\vdots$
Case 11. $X = a_0$

Data values for BACAIC are identified by name when input. All variables are zero initially, and values carry over from one case to the next unless changed. For example, expression 1 means '$I - 1 \rightarrow I$', so the initial value $I = 11$ needs to be input only in case 1.

Expressions 2–4 ensure that the value of expression 2 is the absolute value of X when we get to expression 5. (The '2' in expression 4 means *expression* 2, not the constant 2.) Expression 5 therefore has the value $f(X)$.

A typical way to use BACAIC was to print the values associated with all expressions 1, 2, ...; this was a good way to locate errors. Expression 7 in the program above is an unconditional jump; expression 9 says that the values of I and of expression 5 should be tabulated (printed).

The BACAIC system was easy to learn and to use, but the language was too restrictive for general-purpose computing. One novel feature was its "check-out mode," in which the user furnished hand-calculated data and the machine would print out only the discrepancies it found.

According to [BE 57], BACAIC became operational also on the IBM 650 computer, in August 1956.

Kompilers

Another independent development was taking place almost simultaneously at the University of California Radiation Laboratory in Livermore, California; this work has apparently never been published, except as an internal report [EK 55]. In 1954, A. Kenton Elsworth began to experiment with the translation of algebraic equations into IBM 701

machine language and called his program Kompiler 1; at that time he dealt only with individual formulas, without control statements or constants or input-output. Elsworth and his associates Robert Kuhn, Leona Schloss, and Kenneth Tiede went on to implement a working system named Kompiler 2 during the following year. The latter system was somewhat similar in flavor to ПП-2, except that it was based on flow diagrams instead of operator schemata. They characterized its status in the following way [EK 55, page 4]:

> In many ways Kompiler is an experimental model; it is therefore somewhat limited in applications. For example it is designed to handle only full-word data and is restricted to fixed-point arithmetic. At the same time every effort was made to design a workable and worthwhile routine: the compiled code should approach very closely the efficiency of a hand-tailored code; learning to use it should be relatively easy; compilation itself is very fast.

In order to compensate for the fixed-point arithmetic, special features were included to facilitate scaling. As we shall see, this is perhaps Kompiler 2's most noteworthy aspect.

To solve the TPK problem, let us first agree to scale the numbers by writing

$$A_i = 2^{-10}a_i, \qquad Y = 2^{-10}y, \qquad I = 2^{-35}i.$$

Furthermore we will need to use the scaled constants

$$V = 5 \cdot 2^{-3}, \quad F = 400 \cdot 2^{-10}, \quad N = 999 \cdot 2^{-10}, \quad W = 1 \cdot 2^{-35}.$$

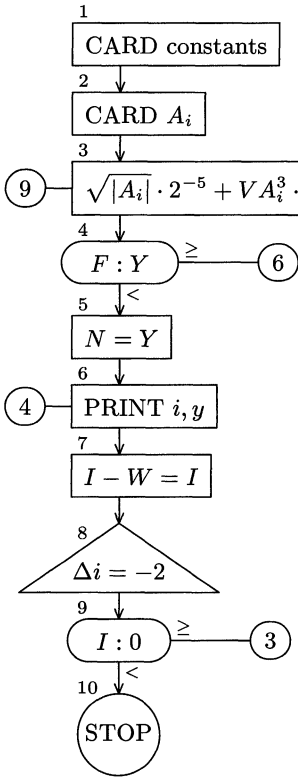
The next step is to draw a special kind of flow diagram for the program, as shown in Figure 4.

The third step is to allocate data storage, for example as follows:

$$61 \equiv I, \quad 63 \equiv Y, \quad 65 \equiv V, \quad 67 \equiv F, \quad 69 \equiv N, \quad 71 \equiv W;$$
$$81 \equiv A_0, \quad 83 \equiv A_1, \quad \ldots, \quad 101 \equiv A_{10}.$$

(Addresses in the IBM 701 go by half-words, but variables in Kompiler 2 occupy full words. Address 61 denotes half-words 60 and 61 in the "second frame" of the memory.)

The final step is to transcribe the flow-diagram information into a fixed format designated for keypunching. The source input to Kompiler 2 has two parts: the "flow-diagram cards," one card per box in the flow diagram; and the "algebraic cards," one per complex equation. In our



Read values of constants and the initial value of I from a data card

Read $A_0, \ldots, A_{10}$ from two more data cards.

Calculate Y.

Go to 6 if $400 \geq y$.

Set y to 999.

Print the answer.

Decrease i by 1.

Decrease the address of A_i by 2 wherever it appears.

Return to 3 if $i \geq 0$.

Stop the machine.

FIGURE 4. This flow diagram for Kompiler input is to be transcribed into numeric format and punched onto cards.

case the flow-diagram cards are

```
 1CARD     61   2     235   0     103 310     310 135       0  61
 2CARD     81   2     310 310     310 310     310 310     310  95
 3CALC    101   8      65         101   8      63
 4TRPL     67  63   6
 5PLUS     69      63
 6PRNT     61  63   2   1  35  10
 7MINS     71  61  61
 8DECR      2
 9TRPL     61   Z   3
10STOP
```

and there are three algebraic cards:

```
1*ΔCARD
2*ΔPRNT
3ΔSRTΔABSA.-05+VA3.+13=Y
```

Here is a free translation of the meaning of the flow diagram cards:

1. Read data cards into locations beginning with 61 in steps of 2. The words of data are to be converted using respective scale codes 235, 0, 103, ..., 0; stop reading cards after the beginning location has become 61, that is, immediately. [The scale code ddbb means to take the 10-digit data as a decimal fraction, multiply it by 10^{dd}, convert to binary, and divide by 2^{bb}. In our case the first input datum will be punched as 1000000000, and the scale code 235 means that this is regarded first as $(10.00000000)_{10}$ and eventually converted to $(.00\ldots01010)_2 = 10 \cdot 2^{-35}$, the initial value of I. The initial value of N, with its scale code 310, would therefore be punched 9990000000. Up to seven words of data are punched per data card.]

2. Read data cards into locations beginning with 81 in steps of 2. The words of data are to be converted using respective scale codes 310, 310, ..., 310; stop reading cards after the beginning location has become 95. The beginning location should advance by 14 between data cards (hence exactly two cards are to be read).

3. Calculate a formula using the variables in the respective locations 101 (which changes at step 8); 65; 101 (which changes at step 8); and 63.

4. If the contents of location 67 minus the contents of location 63 is nonnegative, go to step 6.

5. Store the contents of location 69 in location 63.

6. Print locations 61 through 63, with 2 words per line and 1 line per block. The respective scale factors are 35 and 10.

7. Subtract the contents of location 71 from the contents of location 61 and store the result in location 61.

8. Decrease all locations referring to step 8 by 2 (see step 3).

9. If location 61 contains a nonnegative value (Z stands for zero), go to step 3.

10. Stop the machine.

The first two algebraic cards in our example simply cause the library subroutines for card reading and line printing to be loaded with the object program. The third card is used to encode

$$\sqrt{|A_i|} \cdot 2^{-5} + VA_i^3 \cdot 2^{13} = Y.$$

The variable names on an algebraic card are actually nothing but dummy placeholders, since the storage locations must be specified on the corresponding CALC card. Thus, the third algebraic card could also have been punched as

```
3ΔSRTΔABSX.-05+XX3.+13=X
```

without any effect on the result.

Kompiler 2 was used for several important production programs at Livermore. By 1959 it had been replaced by Kompiler 3, a rather highly developed system for the IBM 704 that used three-line format analogous to the notation of MAC (although it apparently was designed independently).

A Declarative Language

During 1955 and 1956, E. K. Blum at the U.S. Naval Ordnance Laboratory developed a language of a completely different type. His language ADES (Automatic Digital Encoding System) was presented at the ACM national meetings in 1955 (of which no proceedings were published) and in 1956 [BL 56″], and at the ONR symposium in 1956 [BL 56′]. He described it as follows:

> The ADES language is essentially mathematical in structure. It is based on the theory of the recursive functions and the schemata for such functions, as given by Kleene [BL 56′, page 72].
>
> The ADES approach to automatic programming is believed to be entirely new. Mathematically, it has its foundations in the bedrock of the theory of recursive functions. The proposal to apply this theory to automatic programming was first made by C. C. Elgot, a former colleague of the author's. While at the Naval Ordnance Laboratory, Elgot did some research on a language for automatic programming. Some of his ideas were adapted to ADES [BL 56, page iii].

A full description of the language was given in a lengthy report [BL 56]. Several aspects of ADES are rather difficult to understand, and we will content ourselves with a brief glimpse into its structure by considering the following ADES program for TPK. The conventions of [BL 57′] are followed here since they are slightly simpler than the original proposals in [BL 56].

01 $a_0 11 : q_0 11,$
02 $f_{50} = + \sqrt{} \text{ abs } c_1 \cdots 5\ c_1\ c_1\ c_1,$
03 $d_{12} b_1 = r_0,$
04 $d_{22} b_2 = \leq b_3\, 400,\ b_3,\ 999,$
05 $b_3 = f_{50}\, a_0\, r_0,$
06 $r_0 = -\ 10\ q_0,$
07 $\forall\ 0\ q_0\ 10\ b_0 = f_0\ b_1\ b_2,$

Here is a rough translation: Line *01* is the so-called "computer table," meaning that input array a_0 has 11 positions, and that the "independent index symbol" q_0 takes 11 values. Line *02* defines the auxiliary function f_{50}, our $f(t)$; arithmetic expressions were defined in Łukasiewicz's parenthesis-free notation [LU 51, Chapter 4], now commonly known as "left Polish." Variable c_1 here denotes the first parameter of the function. (Incidentally, "right Polish" notation seems to have been first proposed shortly afterwards by C. L. Hamblin in Australia; see [HA 57].)

Line *03* states that the dependent variable b_1 is equal to the dependent index r_0; the 'd_{12}' here means that this quantity is to be output as component 1 of a pair. Line *04* similarly defines b_2, which is to be component 2. This line is a "branch equation" meaning '**if** $b_3 \leq 400$ **then** b_3 **else** 999'. (Such branch equations are an embryonic form of the conditional expressions introduced later by McCarthy into LISP and ALGOL. Blum remarked that the equation '$\leq\ x\ a, f, g,$' could be replaced by $\varphi f + (1-\varphi)g$, where φ is a function that takes the value 1 or 0 according as $x \leq a$ or $x > a$ [BL 56, page 16].)

> The function φ is a primitive recursive function, and could be incorporated into the library as one of the given functions of the system. Nevertheless, the branch equation is included in the language for practical reasons. Many mathematicians are accustomed to that terminology, and it leads to more efficient programs.

In spite of these remarks, Blum may well have intended that f or g not be evaluated or even defined when $\varphi = 0$ or 1, respectively.

Line *05* says that b_3 is the result of applying f_{50} to the r_0th element of a_0. Line *06* explains that r_0 is $10 - q_0$. Finally, line *07* is a so-called "phase equation" that specifies the overall program flow, by saying that b_1 and b_2 are to be evaluated for $q_0 = 0, 1, \ldots, 10$.

The ADES language is "declarative" in the sense that the programmer states relationships between variable quantities without explicitly specifying the order of evaluation. John McCarthy put it this way, in 1958 [ER 58′, page 275]:

> Mathematical notation as it presently exists was developed to facilitate stating mathematical facts, i.e., making declarative sentences. A program gives a machine orders and hence is usually constructed out of imperative sentences. This suggests that it will be necessary to invent new notations for describing complicated procedures, and we will not merely be able to take over

> intact the notations that mathematicians have used for making declarative sentences.

The transcript of a 1965 discussion of declarative versus imperative languages, with comments by P. Abrahams, P. Z. Ingerman, E. T. Irons, P. Naur, B. Raphael, R. V. Smith, C. Strachey, and J. W. Young, appears in *Communications of the ACM* **9** (1966), 155–156, 165–166.

Although ADES was based on recursive function theory, it did not really include recursive procedures in the sense of ALGOL 60; it dealt primarily with special types of recursive equations over the integers, and the emphasis was on studying the memory requirements for evaluating such recurrences.

An experimental version of ADES was implemented on the IBM 650 and described in [BL 57, BL 57′]. Blum's translation scheme was what we now recognize as a recursive approach to the problem, but the recursion was not explicitly stated; he essentially moved things on and off various stacks during the course of the algorithm. This implementation points up the severe problems people had to face in those days: The ADES encoder consisted of 3500 instructions while the Type 650 calculator had room for only 2000, so it was necessary to insert the program card decks into the machine repeatedly, once for each equation! Because of further machine limitations, the program above would have been entered into the computer by punching the following information onto six cards:

```
A00  011  P02  Q00  011  P01  F50  E00  F02  F20
F06  C01  F04  F04  F04  005  C01  C01  C01  P01
D12  B01  E00  R00  P01  D22  B02  E00  F11  B03
400  P01  B03  P01  999  P01  B03  E00  F50  A00
R00  P01  R00  E00  F03  010  Q00  P01  P03  000
Q00  010  B00  E00  F00  B01  B02  P01   -    -
```

Here Pnn was a punctuation mark, Fnn a function code, etc. Actually the implemented version of ADES was a subset that did not allow auxiliary f-equations to be defined, so the definition of b_3 in line *05* would have been written out explicitly.

The IT

> In September, 1955, four members of the Purdue University Computing Laboratory — Mark Koschman, Sylvia Orgel, Alan Perlis, and Joseph W. Smith — began a series of conferences to discuss methods of automatic coding. Joanne Chipps joined the group in March, 1956. A compiler, programmed to be used on the Datatron, was the goal and result [OR 58, page 1].

Purdue received one of the first Datatron computers, manufactured by Electrodata Corporation (see *Journal of the Association for Computing Machinery* **2** (1955), 122, and [PE 55]); this machine was known later as the Burroughs 205. By the summer of 1956, the Purdue group had completed an outline of the basic logic and language of its compiler, and they presented some of their ideas at the ACM national meeting [CK 56]. (This paper, incidentally, used both the words "compiler" and "statement" in the modern sense; a comparison of the ONR 1954 and 1956 symposium proceedings makes it clear that the word "compiler" had by now acquired its new meaning. Furthermore the contemporary FORTRAN manuals [IB 56, IB 57] also used the term "statement" where [IB 54] had said "formula." Terminology was crystallizing.)

At that time Perlis and Smith moved to the Carnegie Institute of Technology, taking copies of the flowcharts with them, and they adapted their language to the IBM 650 (a smaller machine) with the help of Harold Van Zoeren. The compiler was put into use in October 1956 (see [PS 57, page 102]), and it became known as IT, the Internal Translator.

> Compilation proceeds in two phases: 1) translation from an IT program into a symbolic program, PIT and 2) assembly from a PIT program into a specific machine coded program, SPIT [PS 57′, page 1.23].

The intermediate "PIT" program was actually a program in SOAP language [PM 55], the source code for an excellent symbolic assembly program for the IBM 650. Perlis stated that the existence of SOAP was an important simplifying factor in their implementation of IT, which was completed about three months after its authors had learned the 650 machine language.

This software system was the first really *useful* compiler; IT and IT's derivatives were used frequently and successfully in hundreds of computer installations until the 650 became obsolete. (Indeed, R. B. Wise stated in October 1958 that "the IT language is about the closest thing we have today to the universal language among computers" [WA 58, page 131].) The previous systems we have discussed were important steps along the way, but none of them had the combination of powerful language and adequate implementation and documentation needed to make a significant impact in the use of machines. Furthermore, IT proved that useful compilers could be constructed for small computers without enormous investments of manpower.

Here is an IT program for TPK:

```
 1:  READ
 2:  3, I1, 10, −1, 0,
 5:  Y1 ← "20E, AC(I1 + 1)"
                + (5 × (C(I1 + 1) ∗ 3))
 6:  G3 IF 400.0 ≥ Y1
 7:  Y1 ← 999
 3:  TI1 TY1
10:  H
```

Each statement has an identifying number, but the numbers need not be in order. The READ statement does not specify the names of variables being input, since such information appears on the data cards themselves. Floating-point variables are called Y1, Y2, ..., or C1, C2, ...; the example program assumes that the input data will specify eleven values for C1 through C11.

Statement number 2 designates an iteration of the following program through statement number 3 inclusive; variable I1 runs from 10 in steps of −1 down to 0. Statement 5 sets Y1 to $f(\mathrm{C}_{\mathrm{I1}+1})$; the notation "20E, x" is used for "language extension 20 applied to x," where extension 20 happens to be the floating-point square root subroutine. Notice the use of mixed integer and floating-point arithmetic in the second line of statement 5. The redundant parentheses emphasize that IT did not deal with operator precedence, although in this case the parentheses need not have been written since IT evaluated expressions from right to left.

The letter A is used to denote absolute value, and ∗ means exponentiation. Statement 6 goes to 3 if Y1 ≤ 400; and statement 3 outputs I1 and Y1. Statement 10 means "halt."

Since the IBM 650 did not have such a rich character set at the time, the program above would actually be punched onto cards in the following form — using K for comma, M for minus, Q for quotes, L and R for parentheses, etc.:

```
0001  READ                              F
0002  3K I1K 10K M1K 0K                 F
0005  Y1 Z Q 20EK ACLI1S1R Q
0005        S L5 X LCLI1S1R P 3RR       F
0006  G3 IF 400J0 W Y1                  F
0007  Y1 Z 999                          F
0003  TI1 TY1                           F
0010  H                                FF
```

The programmer also supplied a "header card," stating the limits on array subscripts actually used; in this case the header card would specify

one I variable, one Y variable, 11 C variables, 10 statements. An array of statement locations was kept in the running program, since the language allowed programmers to 'go to' statement number n, where n was the value of any integer expression.

The Purdue compiler language discussed in [CK 56] was in some respects richer than this; it included the ability to type out alphabetic information and to define new extensions (functions) in source language. On the other hand, [CK 56] did not mention iteration statements or data input. Joanne Chipps and Sylvia Orgel completed the Datatron implementation in the summer of 1957; the language had lost the richer features in [CK 56], however, probably since they were unexpectedly difficult to implement. Our program would have looked like this in the Purdue compiler language [OR 58]:

input $i0$ $y0$ $c10$ $s10$ f	[maximum subscripts used
1 e "800e" f	[read input]
2 s $i0 = 10$ f	[set $i_0 = 10$]
5 s $y0 =$ "200e, $aci0$" $+ (5 \times (ci0p3))$ f	
6 r $g8$, r $y0 \leq 400.0$ f	[go to 8 if $y_0 \leq 400.0$]
7 s $y0 = 999$ f	
8 o $i0$ f	[output i_0]
9 o $y0$ f	[output y_0]
4 s $i0 = i0 - 1$ f	
3 r $g5$, r $0 \leq i0$ f	[go to 5 if $i_0 \geq 0$]
10 h f	[halt]

Notice that subscripts now may start with 0, and that each statement begins with a letter identifying its type. There are enough differences between this language and IT to make mechanical translation nontrivial.

The Arrival of FORTRAN

During all this time the ongoing work on FORTRAN was widely publicized. Max Goldstein may have summed up the feelings of many people when he made the following remark in June 1956: "As far as automatic programming goes, we have given it some thought and in the scientific spirit we intend to try out FORTRAN when it is available. However ..." [GO 56, page 40].

The day was coming. October 1956 witnessed another "first" in the history of programming languages, namely, the first language description that was carefully written and beautifully typeset, neatly bound with a glossy cover. It began thus [IB 56]:

> This manual supersedes all earlier information about the FORTRAN system. It describes the system which will be made available during late 1956, and is intended to permit planning and FORTRAN coding in advance of that time [page 1].
>
> Object programs produced by FORTRAN will be nearly as efficient as those written by good programmers [page 2].

"Late 1956" was, of course, a euphemism for April 1957. Here is how Saul Rosen described FORTRAN's debut [RO 64, page 4]:

> Like most of the early hardware and software systems, FORTRAN was late in delivery, and didn't really work when it was delivered. At first people thought it would never be done. Then when it was in field test with many bugs, and with some of the most important parts unfinished many thought it would never work. It gradually got to the point where a program in FORTRAN had a reasonable expectancy of compiling all the way through and maybe even of running.

In spite of these difficulties, FORTRAN I was clearly worth waiting for; it soon was accepted even more enthusiastically than its proponents had dreamed [BA 58, page 246]:

> A survey in April of this year [1958] of twenty-six 704 installations indicates that over half of them use FORTRAN for more than half of their problems. Many use it for 80% or more of their work (particularly the newer installations) and almost all use it for some of their work. The latest records of the 704 users' organization, SHARE, show that there are some sixty installations equipped to use FORTRAN (representing 66 machines) and recent reports of usage indicate that more than half the machine instructions for these machines are being produced by FORTRAN.

On the other hand, not everyone had been converted. The second edition of programming's first textbook, by Wilkes, Wheeler, and Gill, was published in 1957, and the authors concluded their newly added chapter on "automatic programming" with the following cautionary remarks [WW 57, pages 136–137]:

> The machine might accept formulas written in ordinary mathematical notation, and punched on a specially designed keyboard perforator. This would appear at first sight to be a very significant development, promising to reduce greatly the labor

> of programming. A number of schemes of formula recognition have been described or proposed, but on examination they are found to be of more limited utility than might have been hoped. ... The best that one could expect a general purpose formula-recognition routine to do, would be to accept a statement of the problem after it had been examined, and if necessary transformed, by a numerical analyst. ... Even in more favorable cases, experienced programmers will be able to obtain greater efficiency by using more conventional methods of programming.

An excellent paper by the authors of FORTRAN I, describing both the language and the organization of the compiler, was presented at the Western Joint Computer Conference in 1957 [BB 57]. The new techniques for global program flow analysis and optimization, due to Robert A. Nelson, Irving Ziller, Lois M. Haibt, and Sheldon Best, were particularly important. By expressing TPK in FORTRAN I we can see most of the changes to FORTRAN 0 that had been adopted:

```
C   THE TPK ALGORITHM, FORTRAN STYLE
    FUNF(T) = SQRTF(ABSF(T))+5.0*T**3
    DIMENSION A(11)
1   FORMAT(6F12.4)
    READ 1, A
    DO 10 J = 1, 11
    I = 11 - J
    Y = FUNF(A(I+1))
    IF (400.0-Y) 4, 8, 8
4   PRINT 5, I
5   FORMAT(I10, 10H TOO LARGE)
    GO TO 10
8   PRINT 9, I, Y
9   FORMAT(I10, F12.7)
10  CONTINUE
    STOP 52525
```

The chief innovations were

1) Provision for comments: No programming language designer had thought to do this before! (Assembly languages had comment cards, but programs in higher-level languages were generally felt to be self-explanatory.)

2) Arithmetic statement functions were introduced. These were not mentioned in [IB 56], but they appeared in [BB 57] and (in detail) in the Programmer's Primer [IB 57, pages 25 and 30–31].

3) Formats were provided for input and output. This feature, due to Roy Nutt, was a major innovation in programming languages; it probably had a significant effect in making FORTRAN popular, since input-output conversions were otherwise very awkward to express on the 704.

4) Lesser features not present in [IB 54] were the `CONTINUE` statement, and the ability to display a five-digit *octal* number when the machine halted at a `STOP` statement.

MATH-MATIC and FLOW-MATIC

Meanwhile, Grace Hopper's programming group at UNIVAC had also been busy. They had begun to develop an algebraic language in 1955, a project that was headed by Charles Katz, and the compiler was released to two installations for experimental tests in 1956 (see [BE 57, page 112]). The language was originally called AT-3; but it received the catchier name MATH-MATIC in April 1957, when its preliminary manual [AB 57] was released. The following program for TPK gives MATH-MATIC's flavor:

```
(1)  READ-ITEM A(11) .
(2)  VARY I 10(-1)0 SENTENCE 3 THRU 10 .
(3)  J = I+1 .
(4)  Y = SQR |A(J)| + 5*A(J)³ .
(5)  IF Y > 400 JUMP TO SENTENCE 8 .
(6)  PRINT-OUT I, Y .
(7)  JUMP TO SENTENCE 10 .
(8)  Z = 999 .
(9)  PRINT-OUT I, Z .
(10) IGNORE .
(11) STOP .
```

The language was quite readable; notice the vertical bar and the superscript 3 in sentence (4), indicating an extended character set that could be used with some peripheral devices. But the MATH-MATIC programmers did *not* share the FORTRAN group's enthusiasm for efficient machine code; they translated MATH-MATIC source language into A-3 (an extension of A-2), and this produced extremely inefficient programs, especially when we consider the fact that arithmetic was all done by floating point subroutines. The UNIVAC computer was no match for an IBM 704 even when it was expertly programmed, so MATH-MATIC was of limited utility.

The other product of Grace Hopper's programming staff was far more influential and successful, since it broke important new ground. This was what she originally called the data processing compiler in January, 1955; it was soon to be known as 'B-0', later as the "Procedure Translator" [KM 57], and finally as FLOW-MATIC [HO 58, TA 60]. The FLOW-MATIC language used English words, somewhat as MATH-MATIC did but more so, and its operations concentrated on business applications. The following examples are typical of FLOW-MATIC operations:

```
(1) COMPARE PART-NUMBER (A) TO PART-NUMBER (B); IF
    GREATER GO TO OPERATION 13; IF EQUAL GO TO
    OPERATION 4; OTHERWISE GO TO OPERATION 2 .
(2) READ-ITEM B; IF END OF DATA GO TO OPERATION 10 .
```

The allowable English templates are shown in [SA 69, pages 317–322].

The first experimental B-0 compiler was operating in 1956 [HO 58, page 171], and it was released to UNIVAC customers in 1958 [SA 69, page 316]. FLOW-MATIC had a significant effect on the design of COBOL in 1959.

A Formula-Controlled Computer

At the international computing colloquium in Dresden, 1955, Klaus Samelson presented the rudiments of a particularly elegant approach to algebraic formula recognition [SA 55], improving on Böhm's technique. Samelson and his colleague F. L. Bauer developed this method during the ensuing years, and their subsequent paper describing it [SB 59] became well known.

One of the first things they did with their approach was to design a computer in which algebraic formulas *themselves* were the machine language. This computer design was submitted to the German patent office in the spring of 1957 [BS 57], and to the U.S. patent office (with the addition of wiring diagrams) a year later. Although the German patent was never granted, and the machines were never actually constructed, Bauer and Samelson eventually received U.S. Patent 3,047,228 for this work [BS 62].

Their patent describes four possible levels of language and machine. At the lowest level they introduced something like the language used on pocket calculators of the 1970s, allowing formulas that consist only of operators, parentheses, and numbers; their highest level included provision for a full-fledged programming language incorporating such features as variables with multiple subscripts, and decimal arithmetic with arbitrary precision.

The language of Bauer and Samelson's highest-level machine is of principal concern to us here. A program for TPK could be entered on its keyboard by typing the following:

01	◇	$0000.00000000 \Rightarrow a{\downarrow}11{\uparrow}$
02		$2.27 \Rightarrow a{\downarrow}1{\uparrow}$
	...	
12		$5.28764 \Rightarrow a{\downarrow}11{\uparrow}$
13		$10 \Rightarrow i$
14	44∗	$a{\downarrow}i{+}1{\uparrow} \Rightarrow t$
15		$\sqrt{\mathrm{B}t + 5 \times t \times t \times t} \Rightarrow y$
16		$i = \square\square \Rightarrow i$
17		$y > 400 \rightarrow 77*$
18		$y = \square\square\square.\square\square\square \Rightarrow y$
19		$\rightarrow 88*$
20	77∗	$999 = \square\square\square \Rightarrow y$
21	88∗	$i - 1 \Rightarrow i$
22		$i > -1 \rightarrow 44*$

(This is the American version; the German version would be the same if all the decimal points were replaced by commas.)

The "◇" at the beginning of this program is optional; it means that the ensuing statements up to the next label (44∗) will not enter the machine's "formula storage," they will simply be performed and forgotten. The remainder of line *01* specifies storage allocation; it says that a is an 11-element array whose entries will contain at most 12 digits.

Lines *02–12* enter the data into array a. The machine also included a paper-tape reader in addition to its keyboard input; and if the data were to be entered from paper tape, lines *02–12* could be replaced by the code

	$1 \Rightarrow i$
33∗	$\bullet\bullet\bullet\bullet\bullet\bullet \Rightarrow a{\downarrow}i{\uparrow}$
	$i + 1 \Rightarrow i$
	$i < 12 \rightarrow 33*$

Actually this input convention was not specifically mentioned in the patent, but Bauer [BA 76′] recalls that such a format was intended.

The symbols ↓ and ↑ for subscripts would be entered on the keyboard but they would not actually appear on the printed page; instead, the printing mechanism was intended to shift up and down. The equal signs followed by square boxes on lines *16*, *18*, and *20* indicate output of a specified number of digits, showing the desired decimal point location. The rest of the program above should be self-explanatory, except perhaps for the B in line *15*, which denotes absolute value (*Betrag*).

Summary

We have now reached the end of our story, having covered essentially every high-level language whose design began before 1957. It is impossible to summarize all of the languages we have discussed by preparing a neat little chart; but everybody likes to see a neat little chart, so here is an attempt at a rough but perhaps meaningful comparison (see Table 1).

Table 1 shows the principal mathematically oriented languages we have discussed, together with their chief authors and approximate year of greatest research or development activity. The "arithmetic" column shows X for languages that deal with integers, F for languages that deal with floating-point numbers, and S for languages that deal with scaled numbers. The remaining columns of Table 1 are filled with highly subjective "ratings" of the languages and associated programming systems according to various criteria.

Implementation: Was the language implemented on a real computer? If so, how efficient and/or easy was it to use?

Readability: How easy is it to read programs in the language? (This aspect includes such things as the variety of symbols usable for variables, and the closeness to familiar notations.)

Control structures: At how high a level are the control structures? Are the existing control structures sufficiently powerful? (By "high level" we mean a level of abstraction — something that the language has but the machine does not.)

Data structures: At how high a level are the data structures? (For example, can variables be subscripted?)

Machine independence: How much does a programmer need to keep in mind about the underlying machine?

Impact: How many people are known to have been directly influenced by this work at the time?

Finally there is a column of "firsts," which states some new thing(s) that this particular language or system introduced.

The Sequel

What have we not seen, among all these languages? The most significant gaps are the lack of high-level *data structures* other than arrays (except in Zuse's unpublished language); the lack of high-level *control structures* other than iteration controlled by an index variable; and the lack of *recursion.* These three concepts, which now are considered absolutely fundamental in computer science, did not find their way into high-level languages until the 1960s. Our languages today probably have too many features, but the languages up to FORTRAN I had too few.

TABLE 1

Language	Principal author(s)	Year	Arith-metic	Imple-men-tation	Read-abil-ity	Control struc-tures	Data struc-tures	Machine indepen-dence	Im-pact	First
Plankalkül	Zuse	1945	X, S, F	F	D	B	A	B	C	Programming language, hierarchic data
Flow diagrams	Goldstine & von Neumann	1946	X, S	F	A	D	C	B	A	Accepted programming methodology
Composition	Curry	1948	X	F	D	C	D	C	F	Code generation algorithm
Short Code	Mauchly	1950	F	C	C	F	F	B	D	High-level language implemented
Intermediate PL	Burks	1950	?	F	A	D	C	A	F	Common subexpression notation
Klammer-ausdrücke	Rutishauser	1951	F	F	B	F	C	B	B	Simple code generation, loop expansion
Formules	Böhm	1951	X	F	B	D	C	B	D	Compiler in own language
AUTOCODE	Glennie	1952	X	C	C	C	C	D	D	Useful compiler
A-2	Hopper	1953	F	C	D	F	F	C	B	Macroexpander
Whirlwind translator	Laning & Zierler	1953	F	B	A	D	C	A	B	Constants in formulas, manual for novices
AUTOCODE	Brooker	1954	X, F	A	B	D	C	A	C	Clean two-level storage
ІПІ-2	Kamynin & Lĭubimskiĭ	1954	F	B	C	D	C	B	D	Code optimization
ІПІ	Ershov	1955	F	B	B	C	C	B	C	Book about a compiler
BACAIC	Grems & Porter	1955	F	A	A	D	F	A	D	Use on two machines
Kompiler 2	Elsworth & Kuhn	1955	S	C	C	D	C	C	F	Scaling aids
PACT I	Working Committee	1955	X, S	A	C	D	C	A	C	Cooperative effort
ADES	Blum	1956	X, F	D	D	B	C	A	F	Declarative language
IT	Perlis	1956	X, F	A	B	C	C	A	A	Successful compiler
FORTRAN I	Backus	1956	X, F	A	A	C	C	A	A	I/O formats, comments, global optimization
MATH-MATIC	Katz	1956	F	B	A	C	C	A	D	Heavy use of English
Patent 3,047,228	Bauer & Samelson	1957	F	D	B	D	C	B	C	Formula-controlled computer

At the time our story leaves off, explosive growth in language development was about to take place, since the successful compilers touched off a language boom. Programming languages had reached a stage when people began to write translators from IT to FORTRAN [GR 58] and from FORTRAN to IT; see [IB 57′] and [BO 58], which describe the FORTRANSIT compiler, developed by Robert W. Bemer, David A. Hemmes, Otto Alexander, Florence H. Pessin, and Leroy May. An excellent survey of the state of automatic programming at the time was prepared by R. W. Bemer [BE 57].

Perhaps the most significant development then in the wind was an international project attempting to define a "standard" algorithmic language. Just after an important meeting in Darmstadt, 1955, a group of European computer scientists began to plan a new language (see [LE 55]), under the auspices of the Gesellschaft für Angewandte Mathematik und Mechanik (GAMM, the Association for Applied Mathematics and Mechanics). They later invited American participation, and an ad hoc ACM committee chaired by Alan Perlis met several times beginning in January 1958. During the summer of that year, Zürich was the site of a meeting attended by representatives of the American and European committees: J. W. Backus, F. L. Bauer, H. Bottenbruch, C. Katz, A. J. Perlis, H. Rutishauser, K. Samelson, and J. H. Wegstein. (See [BB 58] for the language proposed by the European delegates.)

It seems fitting to bring our story to a close by stating the TPK algorithm in the "international algebraic language" (IAL, later called ALGOL 58) developed at that historic Zürich meeting [PS 58]:

```
procedure TPK(a[ ]) =: b[ ];
array (a[0:10], b[0:21]);
comment Given 11 input values a[0],...,a[10], this procedure
          produces 22 output values b[0],...,b[21], according
          to the classical TPK algorithm;
begin for i := 10(−1)0;
  begin  y := f(a[i]);
         f(t) := sqrt(abs(t)) + 5 × t↑3↓;
         if (y > 400); y := 999;
         b[20 − 2 × i] := i;
         b[21 − 2 × i] := y
  end;
  return;
  integer (i)
end TPK
```

The preparation of this paper was supported in part by National Science Foundation Grant MCS 72-03752 A03, by Office of Naval Research contract N00014-76-C-0330, and by IBM Corporation. The authors wish to thank the originators of the languages cited for their many helpful comments on early drafts of this paper.

References

[AB 57] R. Ash, E. Broadwin, V. Della Valle, C. Katz, M. Greene, A. Jenny, and L. Yu, *Preliminary Manual for MATH-MATIC and ARITH-MATIC systems (for Algebraic Translation and Compilation for UNIVAC I and II)* (Philadelphia, Pennsylvania: Remington Rand Univac, 1957), ii + 125 pages.

[AL 54] Charles W. Adams and J. H. Laning, Jr., "The M.I.T. systems of automatic coding: Comprehensive, Summer Session, and Algebraic," *Symposium on Automatic Programming for Digital Computers* (Washington: Office of Naval Research, 1954), 40–68. [Although Laning is listed as co-author, he did not write the paper or attend the conference; in fact, he states that he learned of his "coauthorship" only 10 or 15 years later!]

[BA 54] J. W. Backus, "The IBM 701 Speedcoding system," *Journal of the Association for Computing Machinery* **1** (1954), 4–6.

[BA 58] J. W. Backus, "Automatic programming: Properties and performance of FORTRAN systems I and II," in *Mechanisation of Thought Processes*, National Physical Laboratory Symposium 10 (London: Her Majesty's Stationery Office, 1959), 231–255.

[BA 59] J. W. Backus, "The syntax and semantics of the proposed International Algebraic Language of the Zürich ACM–GAMM conference," in *International Conference on Information Processing*, Proceedings (Paris: UNESCO, 1959), 125–131.

[BA 61] Philip Morrison and Emily Morrison (eds.), *Charles Babbage and his Calculating Engines* (New York: Dover, 1961), xxxviii + 400 pages.

[BA 76] John Backus, "Programming in America in the 1950s — Some personal impressions," in *A History of Computing in the Twentieth Century*, edited by N. Metropolis, J. Howlett, and Gian-Carlo Rota (New York: Academic Press, 1980), 125–135.

[BA 76′] F. L. Bauer, letter to D. E. Knuth (7 July 1976), 2 pages.

[BA 79] John Backus, "The history of FORTRAN I, II, and III," *Annals of the History of Computing* **1** (1979), 21–37. Extended

in *History of Programming Languages*, edited by Richard L. Wexelblat (New York: Academic Press, 1981), 25–73.

[BB 57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The FORTRAN automatic coding system," *Proceedings of the Western Joint Computer Conference* **11** (1957), 188–197.

[BB 58] F. L. Bauer, H. Bottenbruch, H. Rutishauser, and K. Samelson, "Proposal for a universal language for the description of computing processes," in *Computer Programming and Artificial Intelligence*, edited by John W. Carr III (Ann Arbor, Michigan: College of Engineering, University of Michigan, 1958), 353–373. [Translation of an original German draft dated 9 May 1958, in Zürich.]

[BC 54] Arthur W. Burks, Irving M. Copi, and Don W. Warren, *Languages for Analysis of Clerical Problems*, Informal Memorandum 5 (Ann Arbor, Michigan: Engineering Research Institute, University of Michigan, 1954), iii + 24 pages.

[BE 57] R. W. Bemer, "The status of automatic programming for scientific problems," *Computer Applications Symposium* **4** (Chicago, Illinois: Armour Research Foundation, 1957), 107–117.

[BG 53] J. M. Bennett and A. E. Glennie, "Programming for high-speed digital calculating machines," in *Faster Than Thought*, edited by B. V. Bowden (London: Pitman, 1953), 101–113.

[BH 54] John W. Backus and Harlan Herrick, "IBM 701 Speedcoding and other automatic-programming systems," *Symposium on Automatic Programming for Digital Computers* (Washington: Office of Naval Research, 1954), 106–113.

[BH 64] J. W. Backus and W. P. Heising, "FORTRAN," *IEEE Transactions on Electronic Computers* **EC-13** (1964), 382–385.

[BL 56] E. K. Blum, *Automatic Digital Encoding System. II (ADES II)*, NAVORD Report 4209, Aeroballistic Research Report 326 (Washington: U.S. Naval Ordnance Laboratory, 8 February 1956), v + 45 pages plus appendices.

[BL 56′] E. K. Blum, "Automatic Digital Encoding System, II," *Symposium on Advanced Programming Methods for Digital Computers*, ONR Symposium Report ACR-15 (Washington: Office of Naval Research, 1956), 71–76.

[BL 56″] E. K. Blum, "Automatic Digital Encoding System, II (ADES II)," *Proceedings of the ACM National Meeting* **11** (1956), paper 29, 4 pages.

[BL 57] E. K. Blum, *Automatic Digital Encoding System II (ADES II), Part 2: The Encoder*, NAVORD Report 4411 (Washington: U.S. Naval Ordnance Laboratory, 29 November 1956), 82 pages plus appendix.

[BL 57′] E. K. Blum and Shane Stern, *An ADES Encoder for the IBM 650 Calculator*, NAVORD Report 4412 (Washington: U.S. Naval Ordnance Laboratory, 19 December 1956), 15 pages.

[BO 52] Corrado Böhm, "Calculatrices digitales: Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme [Digital computers: On the deciphering of logical-mathematical formulae by the machine itself during the conception of the program]," *Annali di Matematica Pura ed Applicata* (4) **37** (1954), 175–217.

[BO 52′] Corrado Böhm, "Macchina calcolatrice digitale a programma con programma preordinato fisso con tastiera algebrica ridotta atta a comporre formule mediante la combinazione dei singoli elementi simbolici [Programmable digital computer with a fixed preset program and with an algebraic keyboard able to compose formulae by means of the combination of single symbolic elements]," Domanda di brevetto per invenzione industriale [Patent application] No. 13567 di Verbale (Milan: 1 October 1952), 26 pages plus 2 tables.

[BO 54] Corrado Böhm, "Sulla programmazione mediante formule [On programming by means of formulas]," *Atti 4° Sessione Giornate della Scienza*, supplement to *La ricerca scientifica* (Rome: 1954), 1008–1014.

[BO 58] B. C. Borden, "FORTRANSIT, a universal automatic coding system," *Canadian Conference for Computing and Data Processing* (Toronto: University of Toronto Press, 1958), 349–359.

[BP 52] J. M. Bennett, D. G. Prinz, and M. L. Woods, "Interpretative sub-routines," *Proceedings of the ACM National Meeting* (Toronto: 1952), 81–87.

[BR 55] R. A. Brooker, "An attempt to simplify coding for the Manchester electronic computer," *British Journal of Applied Physics* **6** (1955), 307–311. [This paper was received in March 1954.]

[BR 56] R. A. Brooker, "The programming strategy used with the Manchester University Mark I computer," *Proceedings of the Institution of Electrical Engineers* **103**, Supplement, Part B, Convention on Digital Computer Techniques (London: 1956), 151–157.

[BR 58] R. A. Brooker, "The Autocode programs developed for the Manchester University computers," *The Computer Journal* **1** (1958), 15–21.

[BR 58′] R. A. Brooker, "Some technical features of the Manchester Mercury AUTOCODE programme," in *Mechanisation of Thought Processes*, National Physical Laboratory Symposium 10 (London: Her Majesty's Stationery Office, 1959), 201–229.

[BR 60] R. A. Brooker, "MERCURY Autocode: Principles of the program library," *Annual Review in Automatic Programming* **1** (1960), 93–110.

[BS 57] Friedrich Ludwig Bauer and Klaus Samelson, "Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens," *Auslegeschrift 1094019* (Deutsches Patentamt, 30 March 1957, published December 1960), 26 columns plus 6 figures.

[BS 62] Friedrich Ludwig Bauer and Klaus Samelson, "Automatic computing machines and method of operation," *U.S. Patent 3,047,228* (31 July 1962), 32 columns plus 17 figures.

[BU 50] Arthur W. Burks, "The logic of programming electronic digital computers," *Industrial Mathematics* **1** (1950), 36–52.

[BU 51] Arthur W. Burks, *An Intermediate Program Language as an Aid in Program Synthesis*, Report for Burroughs Adding Machine Company (Ann Arbor, Michigan: Engineering Research Institute, University of Michigan, 1951), ii + 15 pages.

[BW 53] R. A. Brooker and D. J. Wheeler, "Floating operations on the EDSAC," *Mathematical Tables and Other Aids to Computation* **7** (1953), 37–47.

[BW 72] F. L. Bauer and H. Wössner, "The 'Plankalkül' of Konrad Zuse: A forerunner of today's programming languages," *Communications of the ACM* **15** (1972), 678–685.

[CH 36] Alonzo Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics* **58** (1936), 345–363.

[CK 56] J. Chipps, M. Koschmann, S. Orgel, A. Perlis, and J. Smith, "A mathematical language compiler," *Proceedings of the ACM National Meeting* **11** (1956), paper 30, 4 pages.

[CL 61] R. F. Clippinger, "FACT — A business compiler: Description and comparison with COBOL and Commercial Translator," *Annual Review in Automatic Programming* **2** (1961), 231–292.

[CU 48] Haskell B. Curry, *On the Composition of Programs for Automatic Computing*, Memorandum 9806 (Silver Spring, Maryland: Naval Ordnance Laboratory, 1949), 52 pages. [Written in July 1948.]

[CU 50] H. B. Curry, *A Program Composition Technique as Applied to Inverse Interpolation*, Memorandum 10337 (Silver Spring, Maryland: Naval Ordnance Laboratory, 1950), 98 pages plus 3 figures.

[CU 50′] H. B. Curry, "The logic of program composition," *Applications Scientifiques de la Logique Mathématique: Actes de 2e Colloque International de Logique Mathématique, Paris, 25–30 Août 1952* (Paris: Gauthier–Villars, 1954), 97–102. [Paper written in March 1950.]

[EK 55] A. Kenton Elsworth, Robert Kuhn, Leona Schloss, and Kenneth Tiede, *Manual for KOMPILER 2*, Report UCRL-4585 (Livermore, California: University of California Radiation Laboratory, 7 November 1955), 66 pages.

[ER 58] A. P. Ershov, *Программирующая Программа для Быстродействующей Электронной Счетной Машины = Programmiruîushchaîa Programma dlîa Bystrodeĭstvuîushcheĭ Elektronnoĭ Schetnoĭ Mashiny* (Moscow: USSR Academy of Sciences, 1958), 116 pages. English translation, *Programming Programme for the BESM Computer*, translated by M. Nadler (London: Pergamon, 1959), v + 158 pages.

[ER 58′] A. P. Ershov, "The work of the Computing Centre of the Academy of Sciences of the USSR in the field of automatic programming," in *Mechanisation of Thought Processes*, National Physical Laboratory Symposium 10 (London: Her Majesty's Stationery Office, 1959), 257–278.

[FE 60] G. E. Felton, "Assembly, interpretive and conversion programs for PEGASUS," *Annual Review in Automatic Programming* **1** (1960), 32–57.

[GL 52] A. E. Glennie, "The automatic coding of an electronic computer," unpublished lecture notes (14 December 1952), 15 pages. [This lecture was delivered at Cambridge University in February 1953.]

[GL 52′] A. E. Glennie, *Automatic Coding*, unpublished manuscript (undated, probably 1952), 18 pages. [This appears to be a draft of a user's manual to be entitled "The Routine AUTOCODE and Its Use."]

[GL 65] Alick E. Glennie, letter to D. E. Knuth (15 September 1965), 6 pages.

[GO 54] Saul Gorn, "Planning universal semi-automatic coding," *Symposium on Automatic Programming for Digital Computers* (Washington: Office of Naval Research, 1954), 74–83.

[GO 56] Max Goldstein, "Computing at Los Alamos, Group T-1," *Symposium on Advanced Programming Methods for Digital Computers*, ONR Symposium Report ACR-15 (Washington: Office of Naval Research, 1956), 39–43.

[GO 57] Saul Gorn, "Standardized programming methods and universal coding," *Journal of the Association for Computing Machinery* **4** (1957), 254–273.

[GO 72] Herman H. Goldstine, *The Computer from Pascal to von Neumann* (Princeton, New Jersey: Princeton University Press, 1972), xi + 378 pages.

[GP 56] Mandalay Grems and R. E. Porter, "A truly automatic computing system," *Proceedings of the Western Joint Computer Conference* **9** (1956), 10–21.

[GR 58] Robert M. Graham, "Translation between algebraic coding languages," *Proceedings of the ACM National Meeting* **13** (1958), paper 29, 2 pages.

[GV 47] Herman H. Goldstine and John von Neumann, *Planning and Coding of Problems for an Electronic Computing Instrument: Report on the Mathematical and Logical Aspects of an Electronic Computing Instrument* (Princeton, New Jersey: Institute for Advanced Study, 1947–1948), Volume 1, iv + 69 pages; Volume 2, iv + 68 pages; Volume 3, iii + 23 pages. Reprinted in von Neumann's *Collected Works* **5**, edited by A. H. Taub (Oxford: Pergamon, 1963), 80–235.

[HA 52] Staff of the Computation Laboratory [Howard H. Aiken and 55 others], "Description of a Magnetic Drum Calculator," *The*

Annals of the Computation Laboratory of Harvard University **25** (Cambridge, Massachusetts: Harvard University Press, 1952), xi + 318 pages.

[HA 57] C. L. Hamblin, "Computer languages," *The Australian Journal of Science* **20** (1957), 135–139.

[HM 53] Grace M. Hopper and John W. Mauchly, "Influence of programming techniques on the design of computers," *Proceedings of the Institute of Radio Engineers* **41** (1953), 1250–1254.

[HO 52] Grace Murray Hopper, "The education of a computer," *Proceedings of the ACM National Meeting* (Pittsburgh, Pennsylvania: 1952), 243–250.

[HO 53] Grace Murray Hopper, "The education of a computer," *Symposium on Industrial Applications of Automatic Computing Equipment* (Kansas City, Missouri: Midwest Research Institute, 1953), 139–144.

[HO 53′] Grace M. Hopper, "Compiling routines," *Computers and Automation* **2**, 4 (May 1953), 1–5.

[HO 55] G. M. Hopper, "Automatic coding for digital computers," *Computers and Automation* **4**, 9 (September 1955), 21–24.

[HO 56] Grace M. Hopper, "The interlude 1954–1956," *Symposium on Advanced Programming Methods for Digital Computers*, ONR Symposium Report ACR-15 (Washington: Office of Naval Research, 1956), 1–2.

[HO 57] Grace M. Hopper, "Automatic programming for business applications," *Computer Applications Symposium* **4** (Chicago, Illinois: Armour Research Foundation, 1957), 45–50.

[HO 58] Grace Murray Hopper, "Automatic programming: Present status and future trends," in *Mechanisation of Thought Processes*, National Physical Laboratory Symposium 10 (London: Her Majesty's Stationery Office, 1959), 155–200.

[HO 71] C. A. R. Hoare, "Proof of a program: FIND," *Communications of the ACM* **14** (1971), 39–45.

[IB 54] Programming Research Group, I.B.M. Applied Science Division, *Specifications for The IBM Mathematical FORmula TRANslating System, FORTRAN*, Preliminary report (New York: I.B.M. Corporation, 1954), i + 29 pages.

[IB 56] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt,

D. Sayre, P. B. Sheridan, H. Stern, and I. Ziller, *Programmer's Reference Manual: The FORTRAN Automatic Coding System for the IBM 704 EDPM* (New York: Applied Science Division and Programming Research Department, I.B.M. Corporation, 15 October 1956), 51 pages.

[IB 57] International Business Machines Corporation, *Programmer's Primer for FORTRAN Automatic Coding System for the IBM 704* (New York: I.B.M. Corporation, 1957), iii + 64 pages.

[IB 57′] International Business Machines Corporation, Applied Programming Department, *FOR TRANSIT: Automatic Coding System for the IBM 650* (New York: I.B.M. Corporation, 1957). See also David A. Hemmes, "FORTRANSIT recollections," *Annals of the History of Computing* **8** (1986), 70–73.

[KA 57] Charles Katz, "Systems of debugging automatic coding," in *Automatic Coding*, Franklin Institute Monograph No. 3 (Lancaster, Pennsylvania: 1957), 17–27.

[KL 58] S. S. Kamynin, E. Z. Lîûbimskiĭ, and M. R. Shura-Bura, "Об автоматизации программирования при помощи программирующей программы = Ob avtomatizatsii programmirovaniîâ pri pomoshchi programmiruîushcheĭ programmy," *Problemy Kibernetiki* **1** (1958), 135–171. English translation, "Automatic programming with a programming programme," *Problems of Cybernetics* **1** (1960), 149–191.

[KM 57] Henry Kinzler and Perry M. Moskowitz, "The procedure translator — A system of automatic programming," in *Automatic Coding*, Franklin Institute Monograph No. 3 (Lancaster, Pennsylvania: 1957), 39–55.

[KN 64] Donald E. Knuth, "Backus Normal Form vs. Backus Naur Form," *Communications of the ACM* **7** (1964), 735–736. [Reprinted as Chapter 2 of the present volume.]

[KN 68] Donald E. Knuth, *Fundamental Algorithms*, Volume 1 of *The Art of Computer Programming* (Reading, Massachusetts: Addison–Wesley, 1968), xxi + 634 pages.

[KN 69] Donald E. Knuth, *Seminumerical Algorithms*, Volume 2 of *The Art of Computer Programming* (Reading, Massachusetts: Addison–Wesley, 1969), xi + 624 pages.

[KN 72] Donald E. Knuth, "Ancient Babylonian algorithms," *Communications of the ACM* **15** (1972), 671–677; **19** (1976), 108.

[Reprinted as Chapter 11 of *Selected Papers on Computer Science*, CSLI Lecture Notes 59 (Stanford, California: Center for the Study of Language and Information, 1996), 185–203.]

[KO 58] L. N. Korolev, "Some methods of automatic coding for BESM and STRELA computers," in *Computer Programming and Artificial Intelligence*, edited by John W. Carr III (Ann Arbor, Michigan: College of Engineering, University of Michigan, 1958), 489–507.

[LA 65] J. H. Laning, letter to D. E. Knuth (13 January 1965), 1 page.

[LA 76] J. H. Laning, letter to D. E. Knuth (2 July 1976), 11 pages.

[LE 55] N. Joachim Lehmann, "Bemerkungen zur Automatisierung der Programmfertigung für Rechenautomaten (Zusammenfassung)," *Elektronische Rechenmaschinen und Informationsverarbeitung*, proceedings of a conference in October 1955 at Darmstadt (Nachrichtentechnische Gesellschaft, 1956), 143. English summary on page 224.

[LJ 58] A. A. Lîapunov, "О логических схемах программ = O logicheskikh skhemakh programm," *Problemy Kibernetiki* **1** (1958), 46–74. English translation, "The logical structure [sic] of programs," *Problems of Cybernetics* **1** (1960), 48–81.

[LM 70] J. Halcombe Laning and James S. Miller, *The MAC Algebraic Language*, Report R-681 (Cambridge, Massachusetts: Instrumentation Laboratory, Massachusetts Institute of Technology, November 1970), 23 pages.

[LU 51] Jan Łukasiewicz, *Aristotle's Syllogistic from the Standpoint of Modern Formal Logic* (Oxford: Clarendon Press, 1951), xii + 141 pages.

[LZ 54] J. H. Laning Jr. and N. Zierler, *A Program for Translation of Mathematical Equations for Whirlwind I*, Engineering Memorandum E-364 (Cambridge, Massachusetts: Instrumentation Laboratory, Massachusetts Institute of Technology, January 1954), v + 21 pages.

[MG 53] E. N. Mutch and S. Gill, "Conversion routines," in *Automatic Digital Computation*, Proceedings of a symposium at the National Physical Laboratory on 25–28 March 1953 (London: Her Majesty's Stationery Office, 1954), 74–80.

[MO 54] Nora B. Moser, "Compiler method of automatic programming," *Symposium on Automatic Programming for Digital Computers* (Washington: Office of Naval Research, 1954), 15–21.

[NA 54] U.S. Navy Mathematical Computing Advisory Panel, *Symposium on Automatic Programming for Digital Computers* (Washington: Office of Naval Research, 1954), v + 152 pages.

[OR 58] Sylvia Orgel, *Purdue Compiler: General Description* (West Lafayette, Indiana: Purdue Research Foundation, 1958), iv+33 pages.

[PE 55] A. J. Perlis, "DATATRON," transcript of a lecture given 11 August 1955, in *Digital Computers and Data Processors*, edited by J. W. Carr III and N. R. Scott (Ann Arbor, Michigan: College of Engineering, University of Michigan, 1956), Section VII.20.1.

[PE 57] Richard M. Petersen, "Automatic coding at G.E.," in *Automatic Coding*, Franklin Institute Monograph No. 3 (Lancaster, Pennsylvania: 1957), 3–16.

[PM 55] Stanley Poley and Grace Mitchell, *Symbolic Optimum Assembly Programming (SOAP)*, 650 Programming Bulletin 1, Form 22-6285-1 (New York: IBM Corporation, November 1955), 4 pages.

[PR 55] Programming Research Section, Eckert Mauchly Division, Remington Rand, "Automatic programming: The A-2 Compiler System," *Computers and Automation* **4**, 9 (September 1955), 25–29; **4**, 10 (October 1955), 15–27.

[PS 57] A. J. Perlis and J. W. Smith, "A mathematical language compiler," in *Automatic Coding*, Franklin Institute Monograph No. 3 (Lancaster, Pennsylvania: 1957), 87–102.

[PS 57′] A. J. Perlis, J. W. Smith, and H. R. Van Zoeren, *Internal Translator (IT): A Compiler for the 650* (Pittsburgh, Pennsylvania: Computation Center, Carnegie Institute of Technology, March 1957), iv + 47 + 68 + 12 pages. Part I, Programmer's Guide; Part II, Program Analysis (the complete source code listing); Part III, Addenda; flow charts were promised on page 3.12, but they may never have been completed. Reprinted in *Applications of Logic to Advanced Digital Computer Programming* (Ann Arbor, Michigan: College of Engineering, University of Michigan, 1957); this report was also available from IBM Corporation as a 650 Library Program, File Number 2.1.001. [Autobiographical note: D. E. Knuth learned about system programming by reading the program listings of Part II in the summer of 1957; this experience changed his life.]

[PS 58] A. J. Perlis and K. Samelson, "Preliminary report, International Algebraic Language," *Communications of the ACM* **1**, 12 (December 1958), 8–22. Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, *Numerische Mathematik* **1** (1959), 41–60. Reprinted in *Annual Review in Automatic Programming* **1** (1960), 268–290.

[RA 73] Brian Randell, *The Origins of Digital Computers: Selected Papers* (Berlin: Springer, 1973), xvi + 464 pages.

[RO 52] Nathaniel Rochester, "Symbolic programming," *IRE Transactions on Electronic Computers* **EC-2**, 1 (March 1953), 10–15.

[RO 64] Saul Rosen, "Programming systems and languages, a historical survey," *Proceedings of the Spring Joint Computer Conference* **25** (1964), 1–16.

[RR 53] Remington Rand, Inc., *The A-2 Compiler System Operations Manual* (15 November 1953), ii + 54 pages. (Prepared by Richard K. Ridgway and Margaret H. Harper under the direction of Grace M. Hopper.)

[RR 55] Remington Rand UNIVAC, *UNIVAC Short Code*, an unpublished collection of dittoed notes. Preface by A. B. Tonik (25 October 1955), 1 page; preface by J. R. Logan (undated but apparently from 1952), 1 page; Preliminary Exposition (1952?), 22 pages, where pages 20–22 appear to be a later replacement; Short Code Supplementary Information, Topic One, 7 pages; Addenda #1–4, 9 pages.

[RU 52] Heinz Rutishauser, *Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen [Automatic Machine-Code Generation on Program-Directed Computers]*, Mitteilungen aus dem Institut für Angewandte Mathematik an der ETH Zürich, No. 3 (Basel: Birkhäuser, 1952), ii + 45 pages.

[RU 55] Heinz Rutishauser, "Some programming techniques for the ERMETH," *Journal of the Association for Computing Machinery* **2** (1955), 1–4.

[RU 55′] H. Rutishauser, "Maßnahmen zur Vereinfachung des Programmierens (Bericht über die in fünfjähriger Programmierungsarbeit mit der Z4 gewonnenen Erfahrungen)," *Elektronische Rechenmaschinen und Informationsverarbeitung*, proceedings

of a conference in October 1955 at Darmstadt (Nachrichtentechnische Gesellschaft, 1956), 26–30. English summary on page 225.

[RU 61] H. Rutishauser, "Interference with an ALGOL procedure," *Annual Review in Automatic Programming* **2** (1961), 67–76.

[RU 63] H. Rutishauser, letter to D. E. Knuth (11 October 1963), 2 pages.

[SA 55] Klaus Samelson, "Probleme der Programmierungstechnik," in *Aktuelle Probleme der Rechentechnik*, edited by N. Joachim Lehmann, Bericht über das Internationale Mathematiker-Kolloquium, Dresden, 22–25 November 1955 (Berlin: Deutscher Verlag der Wissenschaften, 1957), 61–68.

[SA 69] Jean E. Sammet, *Programming Languages: History and Fundamentals* (Englewood Cliffs, New Jersey: Prentice–Hall, 1969), xxx + 785 pages.

[SB 59] K. Samelson and F. L. Bauer, "Sequentielle Formelübersetzung," *Elektronische Rechenanlagen* **1** (1959), 176–182; "Sequential formula translation," *Communications of the ACM* **3** (1960), 76–83, 351.

[SM 73] Leland Smith, "Editing and printing music by computer," *Journal of Music Theory* **17** (1973), 292–309.

[ST 52] C. S. Strachey, "Logical or non-mathematical programmes," *Proceedings of the ACM National Meeting* (Toronto: 1952), 46–49.

[TA 56] D. Tamari, review of [BO 52], *Zentralblatt für Mathematik und ihre Grenzgebiete* **57** (1956), 107–108.

[TA 60] Alan E. Taylor, "The FLOW-MATIC and MATH-MATIC automatic programming systems," *Annual Review in Automatic Programming* **1** (1960), 196–206.

[TH 55] Bruno Thüring, "Die UNIVAC A-2 Compiler Methode der automatischen Programmierung," *Elektronische Rechenmaschinen und Informationsverarbeitung*, proceedings of a conference in October 1955 at Darmstadt (Nachrichtentechnische Gesellschaft, 1956), 154–156. English summary on page 226.

[TU 36] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society* (2) **42** (1936), 230–265; **43** (1937), 544–546.

[WA 54] John Waite, "Editing generators," *Symposium on Automatic Programming for Digital Computers* (Washington: Office of Naval Research, 1954), 22–29.

[WA 58] F. Way III, "Current developments in computer programming techniques," *Computer Applications Symposium* **5** (Chicago, Illinois: Armour Research Foundation, 1958), 125–132.

[WH 50] D. J. Wheeler, "Programme organization and initial orders for the EDSAC," *Proceedings of the Royal Society* **A202** (1950), 573–589.

[WI 52] M. V. Wilkes, "Pure and applied programming," *Proceedings of the ACM National Meeting* (Toronto: 1952), 121–124.

[WI 53] M. V. Wilkes, "The use of a 'floating address' system for orders in an automatic digital computer," *Proceedings of the Cambridge Philosophical Society* **49** (1953), 84–89.

[WO 51] M. Woodger, "A comparison of one and three address codes," *Manchester University Computer, Inaugural Conference* (Manchester: 1951), 19–23.

[WR 71] W. A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS, a language for systems programming," *Communications of the ACM* **14** (1971), 780–790.

[WW 51] Maurice V. Wilkes, David J. Wheeler, and Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer:* With special reference to the EDSAC and the use of a library of subroutines (Cambridge, Massachusetts: Addison–Wesley, 1951), xi + 170 pages. Reprinted, with an introduction by Martin Campbell-Kelly, as Volume 1 in the Charles Babbage Institute Reprint Series for the History of Computing (Los Angeles: Tomash, 1982).

[WW 57] Maurice V. Wilkes, David J. Wheeler, and Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer*, second edition (Reading, Massachusetts: Addison–Wesley, 1957), xii + 238 pages.

[ZU 44] K. Zuse, "Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen [Beginnings of a theory of calculation in general, considering in particular the propositional calculus and its application to relay circuits]," manuscript prepared in 1944. Chapter 1 has been published

in *Berichte der Gesellschaft für Mathematik und Datenverarbeitung* **63** (1972), Part 1, 32 pages. English translation, **106** (1976), 7–20.

[ZU 45] K. Zuse, *Der Plankalkül*, manuscript prepared in 1945. Published in *Berichte der Gesellschaft für Mathematik und Datenverarbeitung* **63** (1972), Part 3, 285 pages. English translation of all but pages 176–196, **106** (1976), 42–244.

[ZU 48] K. Zuse, "Über den Allgemeinen [sic] Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben," *Archiv der Mathematik* **1** (1949), 441–449.

[ZU 59] K. Zuse, "Über den Plankalkül," *Elektronische Rechenanlagen* **1** (1959), 68–71.

[ZU 72] Konrad Zuse, "Kommentar zum Plankalkül," *Berichte der Gesellschaft für Mathematik und Datenverarbeitung* **63** (1972), Part 2, 36 pages. English translation, **106** (1976), 21–41.

Addendum

Another language, PACT I, deserves to be part of the story as well, so it has been included in Table 1 above although it was unfortunately missed by the authors when we first compiled this history. The "Project for the Advancement of Coding Techniques" was a joint effort between programmers from many different installations of IBM 701 computers in Southern California, as they sought ways to make their programs more easily coded, debugged, and portable to future machines without introducing run-time inefficiency.

Their solution was to introduce a nearly machine-independent language with one step per card and automatic scaling of fixed-point arithmetic. For example, TPK would look like this:

Region	*Step*	*Op*	*Clue*	*Factor*	S_1	S_2	
F	1			X			[Take X]
F	2	ABS					[Compute the absolute value]
F	3	SQRT					[Compute the square root]
F	10			X			[Take X]
F	11	X					[Multiply by itself]
F	12	X		X			[Multiply by X]
F	13	X		5			[Multiply by 5]
F	20	+	R	3			[Add the result of step 3]
F	30	EQ		Y			[Store in Y]

```
TPK     0  READ                                    [Read all data cards]
TPK     2                    11                   [Take 11]
TPK     3  EQ                II                   [Store in II]
TPK     4  SET                        I    1      [Begin loop, with I = 1]
TPK     6  USE                        J    II     [Begin nonloop, with J = II]
TPK    10                     A      J           [Take A_J]
TPK    11  EQ                 X                  [Store in X]
TPK    12  DO                 F                  [Perform F]
TPK    15            N        Y                  [Take the negative of Y]
TPK    16  +                400                  [Add 400]
TPK    17  TP              25                     [Transfer to step 25 if ≥ 0]
TPK    18                  999                    [Take 999]
TPK    19  EQ                 Y                  [Store in Y]
TPK    25  LIST                                  [Print II and Y]
TPK    26  ID                 II
TPK    27  ID                 Y
TPK    30                     II                 [Take II]
TPK    31  -                  1                  [Subtract 1]
TPK    32  EQ                 II                 [Store in II]
TPK    33  TEST                       II   11     [I ← I+1, repeat loop if ≤ 11]
TPK    40  HALT                                  [Stop]

Variable   Q   D1   D2
       X  25                [X has 25 fraction bits]
       Y  25                [Y has 25 fraction bits]
      II   0                [II is an integer]
       A  25   11           [A1, ..., A11 have 25 fraction bits]
```

Details can be found in a series of seven articles by some of the principal contributors: Wesley S. Melahn, "A description of a cooperative venture in the production of an automatic coding system," *Journal of the Association for Computing Machinery* **3** (1956), 266–271; Charles L. Baker, "The PACT I coding system for the IBM Type 701," *Journal of the Association for Computing Machinery* **3** (1956), 272–278; Owen R. Mock, "Logical organization of the PACT I compiler," *Journal of the Association for Computing Machinery* **3** (1956), 279–287; Robert C. Miller, Jr., and Bruce G. Oldfield, "Producing computer instructions for the PACT I compiler," *Journal of the Association for Computing Machinery* **3** (1956), 288–291; Gus Hempstead and Jules I. Schwartz, "PACT loop expansion," *Journal of the Association for Computing Machinery* **3** (1956), 292–298; J. I. Derr and R. C. Luke, "Semi-automatic allocation of data storage for PACT I," *Journal of the Association for Computing Machinery* **3** (1956), 299–308; I. D. Greenwald and H. G. Martin, "Conclusions after using the PACT I advanced coding technique," *Journal of the Association for Computing Machinery* **3** (1956), 309–313.

> This project was accomplished through joint effort on the part of many companies where competition is extremely high and almost all discoveries and techniques are considered proprietary information. In the midst of this highly competitive industry, a group of far-sighted men in charge of computing groups realized a need for a better coding technique and established what is now known as the Project for the Advancement of Coding Techniques. The formation of the project was unique, and, we feel, the result of the project is also unique. And it is our sincere desire that the spirit of cooperation exemplified here will continue and possibly prove contagious.

One of the first fruits of this cooperation was the founding late in 1955 of SHARE, the first computer users' organization. See Paul Armer, "SHARE, a eulogy to cooperative effort," *Annals of the History of Computing* **2** (1980), 122–129, the transcript of a lecture given in 1956; Fred J. Gruenberger, "A short history of digital computing in Southern California," *Annals of the History of Computing* **2** (1980), 246–250.

An interesting article by Martin Campbell-Kelly, "Programming the EDSAC: Early programming activity at the University of Cambridge," *Annals of the History of Computing* **2** (1980), 7–36 [reprinted in *IEEE Annals of the History of Computing* **20**, 4 (October–December 1998), 46–67] reconstructs how the TPK algorithm might have been programmed on the EDSAC 1 in late 1949 or early 1950. David J. Wheeler has also explained how TPK would have appeared in 1958 on EDSAC 2, in an appendix to his paper "The EDSAC programming systems," *IEEE Annals of the History of Computing* **14**, 4 (1992), 34–40.

Chapter 2

Backus Normal Form versus Backus Naur Form

[Originally published in Communications of the ACM **7** *(1964), 735–736.]*

Dear Editor:

In recent years it has become customary to refer to syntax presented in the manner of the ALGOL 60 report as "Backus Normal Form." I am not sure where this terminology originated; personally I first recall reading it in a survey article by S. Gorn [2].

Several of us working in the field have never cared for the name Backus Normal Form because it isn't a "Normal Form" in the conventional sense. A normal form usually refers to some sort of special representation that is not necessarily a canonical form.

For example, any ALGOL-60-like syntax can readily be transformed so that all definitions except the definition of ⟨empty⟩ have one of the three forms

(i) ⟨A⟩ ::= ⟨B⟩ | ⟨C⟩, (ii) ⟨A⟩ ::= ⟨B⟩⟨C⟩, or (iii) ⟨A⟩ ::= a.

A syntax in which all definitions are of this kind may be said to be in "Floyd Normal Form," since this point was first raised in a note by R. W. Floyd [1]. But I hasten to withdraw such a term from further use, since other people have doubtless used that simple fact independently in their own work, and the point is only incidental to the main considerations of Floyd's note.

Many people have objected to the term Backus Normal Form because it is just a new name for an old concept in linguistics: An equivalent type of syntax has been used under various other names (Chomsky type 2 grammar, simple phrase structure grammar, context free grammar, and so forth).

There is still a reason for distinguishing between the names, however, since linguists present the syntax in the form of productions while the Backus version has a quite different *form.* (It is a Form for a syntax,

not a Normal Form.) The five principal things that distinguish Backus's form from production form are:

i) Nonterminal symbols are distinguished from terminal letters by enclosing them in special brackets.

ii) All alternatives for a definition are grouped together. (That is, in a production system the three productions 'A $\rightarrow$ BC, A $\rightarrow$ d, A $\rightarrow$ C' would all be written separately, instead of combining them in the single rule '$\langle$A$\rangle$ ::= $\langle$B$\rangle\langle$C$\rangle$ | d | $\langle$C$\rangle$'.)

iii) The symbol '::=' is used to separate left from right.

iv) The symbol '|' is used to separate alternatives.

v) Full names, indicating the meaning of the strings being defined, are used for nonterminal symbols.

Of these five items, (iii) is clearly irrelevant and the peculiar symbol '::=' can be replaced by anything desired; '$\rightarrow$' is perhaps better, to correspond more closely with productions.

But (i), (ii), (iv), and (v) are each important for the *explanatory power* of a syntax. It is quite difficult to fathom the significance of a language defined by productions, compared to the documentation afforded by a syntax that incorporates (i), (ii), (iv), and (v).

On the other hand, it is much easier to do *theoretical* manipulations by using production systems and systematically *avoiding* (i), (ii), (iv), and (v).

For these reasons, Backus's form deserves a special distinguishing name.

Actually, however, only (i) and (ii) were really used by John Backus when he proposed his notation; (iii), (iv), and (v) are due to Peter Naur, who incorporated these changes when drafting the ALGOL 60 report. Naur's additions, particularly (v), are quite important.

Furthermore, if it had not been for Naur's work in recognizing the potential of Backus's ideas and popularizing them with the ALGOL committee, Backus's work would have become virtually lost; and much of the knowledge we have today about languages and compilers would not have been acquired.

Therefore I propose that henceforth we always say "*Backus Naur Form*" instead of Backus Normal Form, when referring to such a syntax. This terminology has several advantages: (1) It gives the proper credit to both Backus and Naur. (2) It preserves the oft-used abbreviation "BNF". (3) It does not call a Form a Normal Form.

I have been saying Backus Naur Form for about two months now and am still quite pleased with it, so I think perhaps everyone else will enjoy this term also.

Donald E. Knuth
California Institute of Technology
Pasadena, California

References

[1] Robert W. Floyd, "Note on mathematical induction in phrase structure grammars," *Information and Control* **4** (1961), 353–358.

[2] Saul Gorn, "Specification languages for mechanical languages and their processors — a baker's dozen," *Communications of the ACM* **4** (1961), 532–542.

Addendum

The proposed change has been widely adopted — and with an en-dash, which makes it even better. For example, the standard reference manual for the Ada programming language [3, §1.1.4] says that Ada's context-free syntax "is described using a simple variant of Backus–Naur form." Michael Woodger has noted [5] that

> One of the key modifications made by Naur to Backus' notation was to choose the designations of the syntactic constituents, such as the "basic statement," to be exactly the same as those used in describing the semantics, without abbreviation. This is what makes the syntax readable, rather than just a useful mathematical notation, and is largely responsible for the success of "BNF" as we know it. Peter Naur was too modest to say this himself in the appendix to his paper [4].

[3] *Consolidated Ada Reference Manual: Language and Standard Libraries*, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1, edited by S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, and Erhard Ploedereder (Berlin: Springer, 2001).

[4] Peter Naur, "The European side of the last phase of the development of ALGOL 60," in *History of Programming Languages*, edited by Richard L. Wexelblat (New York: Academic Press, 1981), 92–139, 147–170.

[5] Mike Woodger, "What does BNF stand for?" *Annals of the History of Computing* **12** (1990), 71–72.

Chapter 3

Teaching ALGOL 60

[Originally published in ALGOL Bulletin **19** *(Amsterdam: Mathematisch Centrum, January 1965), 4–6.]*

Dear Editor:

Many readers of the *ALGOL Bulletin* are directly or indirectly concerned with teaching the ALGOL 60 language. The following two exercises and/or test problems have proved to be especially instructive in the ALGOL classes that I have been teaching intermittently since the spring of 1961, so I feel they will be of interest.

Problem 1. Find all ways to write an unsigned number meaning "one," using at most five basic symbols. The ALGOL report defines unsigned numbers as follows:

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩⟨digit⟩

⟨integer⟩ ::= ⟨unsigned integer⟩ | +⟨unsigned integer⟩
| −⟨unsigned integer⟩

⟨decimal fraction⟩ ::= .⟨unsigned integer⟩

⟨exponent part⟩ ::= $_{10}$⟨integer⟩

⟨decimal number⟩ ::= ⟨unsigned integer⟩ | ⟨decimal fraction⟩
| ⟨unsigned integer⟩⟨decimal fraction⟩

⟨unsigned number⟩ ::= ⟨decimal number⟩ | ⟨exponent part⟩
| ⟨decimal number⟩⟨exponent part⟩

Examples: 1, 1.0, $001_{10}0$, etc.; notice that '$_{10}$' is considered to be a single basic symbol. A prize will be given for any paper with 42 or more correct answers.

Problem 2. When I tried to run the following program, the compiler did not accept it. What errors have I made? [*Note:* The compiler reports

that there are at least 20 bad mistakes, and it makes several other minor objections that are not clear-cut errors. You are only supposed to find errors that are specifically outlawed in the ALGOL 60 language.]

```
begin real I; I := 5;
  begin array M[1:I];
    integer I, J, K;
    real X, Y, Z;
    Boolean P, Q, R;
    switch S = L1;
    real procedure A(B, C);
      integer B;
      label C;
      value B;
      begin integer R;
        X := (B ↑ −2) ↑ (−2);
        go to if P then S[B] else L2;
      L1: Y := P;
        if M(5) = X ÷ Y then
          for B = 1 step 0 until A(B, C) do
          L2: comment now the fun begins;
            begin own array N[if R = true then 2:10 else 1:10
              if X ≠ 0 then
                if Y = 1 or 2 then go to C
            end;
        return
      end procedure A;
    X := I := 0;
    if X = 0 then
      for Q := true, false do
      L3: Z = A(B) the 2nd parameter is: (L3)
    else
  end
end of program;
```

Remarks. Problem 1 can be used very early in a training course. It teaches not only the admissible forms of constants, but also gives important instruction on the use of syntax. Readers may convince themselves that there are precisely 41 correct solutions; exactly (11, 0, 9, 12, 6, and 3) of them have an exponent part of length (0, 1, 2, 3, 4, and 5), respectively. The solution most often missed is perhaps '$.01_{10}2$'.

Problem 2 is suitable for a final exam. This problem stresses the errors that are most commonly made, as well as a few subtle ones and a few obvious ones (to make the problem more interesting). The following errors are considered most important:

1. **array** M: No bound pairs may use variables not declared in an external block. The identifier I is declared in the present block, so its scope is clear.
2. **switch** S: '=' should be ':='.
3. *L1* is undeclared in the block where it has been used.
4. The **value** part should come before the specification part.
5. '$B \uparrow -2$' is not well formed according to ALGOL's syntax.
6. **else** *L2*: We aren't allowed to **go to** *L2* from outside the **for** statement in which *L2* is a statement label.
7. $Y := P$: Only Boolean variables can receive Boolean values.
8. '$M(5)$' should be '$M[5]$'.
9. $X \div Y$ is improper because X and Y are **real**.
10. '**for** $B =$ ' should be '**for** $B :=$ '.
11. **comment** must follow ';' or '**begin**'.
12. '**own array**' should be '**own real array**' or have some other type.
13. '2:10 **else** 1:10' should be, for example, '2 **else** 1 : 10'.
14. '**if** $Y = 1$ **or** 2' is COBOL, not ALGOL. The programmer presumably meant to say '**if** $Y = 1 \vee Y = 2$'.
15. '**then if**' is syntactically wrong, since no conditional statement may follow an **if** clause.
16. '**return**' is an undeclared procedure, or something left over from ALGOL 58.
17. $X := I := 0$: In a multiple assignment, variables must have the same type.
18. **for** $Q :=$ **true**, **false do**: Boolean expressions aren't allowed as elements of a **for** list.
19. $Z =:$ Again, '=' should be ':='.
20. $A(B)$: B is not a declared identifier in this block.
21. 'the 2nd parameter' is illegal because 2 is not a letter; only letter strings are allowed within a parameter delimiter.
22. '**else**' should never complete '**then**' when a **for** statement intervenes.

23. A is a real procedure, but no assignment '$A :=$ value' appears.
24. The final semicolon does not belong there, according to the syntax for ⟨program⟩ and the comment conventions.
25. Many variables are used before any assignment of a value has been given.
26. '$R =$ **true**' is not well formed, since R has been redeclared to have integer type.

Problem 2 is also of possible interest for authors of compiler error-detection procedures.

Donald E. Knuth
Assistant Professor of Mathematics
California Institute of Technology
Pasadena, California, USA

Chapter 4

ALGOL 60 *Confidential*

[Written with Jack N. Merner. Originally published in Communications of the ACM **4** *(1961), 268–272.]*

I would question the ability of anybody who uses that feature.
—H. D. Huskey

Although permitted, ... this will clearly not ordinarily be used.
—H. C. Thatcher, Jr.

The ALGOL 60 Report [5], when first encountered, seems to describe a very complex language that will be difficult to learn. Its "metalinguistic formulae" admirably serve the purpose of specifying a language precisely, but they are certainly not very readable for a beginner.

Experience has shown, however, that ALGOL is in fact easily approachable, once the report is explained, and that ALGOL programs are easy to write. The language is so general and powerful, it can handle an enormous class of problems. The parts of ALGOL that are present in other compiler languages are particularly simple: One quickly learns how to write assignment and **go to** and **for** statements, etc. Indeed, a lot of the unnecessary restrictions imposed by other languages have finally been lifted.

But ALGOL also allows many nonobvious things to be written, and herein lies a problem: ALGOL seems to have become too general. So many restrictions have been lifted that a lot of technical details crop up; many of its permissible constructions are hard to understand and to use correctly. In this paper some of the more obscure features of the language are considered and their usefulness is discussed. Remarks are based on the authors' interpretations of the ALGOL 60 Report.

1. Types

The expression $2 \uparrow X$, where X has been declared to be an integer variable, has either real or integer type depending on whether the value

of X at running time is negative or not. This rule makes it unnecessarily difficult, on most machines,* for a compiler to decide whether to use a floating-point instruction or a fixed-point instruction, and it causes inefficient output. There seems to be no need for such generality.

2. Expressions

A procedure call within an expression might change the value of other components of the expression. For example, suppose that in $F(X, Y)$ the real procedure F stores $\cos(X)$ in Y, and that the value of the procedure is $\sin(X)$. Such a procedure is quite reasonable because people often wish to calculate both the sine and cosine of the same angle. But what then is the meaning of $F(X, Y) + Y$? Is it different from $Y + F(X, Y)$?

A strict interpretation of ALGOL would seem to require that such addition would be noncommutative, for "the sequence of operations is generally from left to right." In other words, the machine program for $F(X, Y) + Y$ would have the form

```
(CALL   F(X,Y))
 FLOAT-ADD  Y
```

while the corresponding left-to-right program for $Y + F(X, Y)$ would be

```
 LOAD       Y
 STORE      TEMP
(CALL   F(X,Y))
 FLOAT-ADD  TEMP
```

A question then arises on how to evaluate $Y + F(X, Y) \uparrow 2$. We are told to perform exponentiation before addition; so one might argue that Y should be changed before the addition. But, looking at this example together with the previous one, we see that a '+' operation is to be performed; so the left-to-right rule tells us to first evaluate its left operand Y, *then* evaluate its right operand $F(X, Y) \uparrow 2$, then perform the addition.

Furthermore, consider the Boolean expression

$$W = 0 \ \wedge \ F(X, Y) = 0$$

with respect to the same procedure F. If W is unequal to zero, must we evaluate $F(X, Y)$? We know that the value of the Boolean expression

---

* The Burroughs B5000 computer will handle integer and real values with the same operation code, but traditional machines do not.

will be false; but does Y get set to $\cos(X)$ or not? A strict interpretation of [5] would evaluate $F(X, Y)$ even when W is nonzero.

Of what value is this extra generality, when it actually causes machine implementations to be less efficient and the language to be more difficult to comprehend?

At the other extreme, there is a restriction in the formation of expressions that seems to be totally unnecessary. We are forbidden from writing $A\uparrow -2$, $X/-Y$, and so on, although such expressions are unambiguous and quite common in mathematical usage (and easy to translate into machine language).

3. Unfinished Expressions

Suppose $G(X, L)$ is a real procedure for which L is a label parameter. Can we write, for example, $Y := G(X, L)$ and then allow procedure G to '**go to** L'? Although this case is not mentioned explicitly in the ALGOL Report, we believe it is not well-formed, since function designators are supposed to "define single numerical or logical values." If G exits to L, it does not define any value. Thus there seems to be no use for a designational expression as an actual parameter to a function-like procedure.

4. Labels

What good are numeric labels? In the designational expression

$$\textbf{if } Ab < c \textbf{ then } 17 \textbf{ else } q[\textbf{if } w < 0 \textbf{ then } 2 \textbf{ else } n]$$

(which occurs at least twice in the ALGOL Report) one has to look closely to discover that 17 is a label, but 2 is not.

Programmers can get into trouble using numeric labels. Consider

$$\textbf{procedure } a(b);\ \textbf{procedure } b;\ 2{:}\ b(2)$$

for example. In this procedure declaration, is 2 supposed to mean the label 2 or the number 2? As a matter of fact, the procedure parameter b might want 2 to be a label in some calls of a and a numeric value in other calls, perhaps *both* in others. Such behavior can be implemented in machine language, but is there a practical reason to do so?

It is gratifying to see that some machine implementations of ALGOL 60 (in particular Irons's PSYCO for the 1604, and Naur's DASK ALGOL [6] for the DASK computer) have ruled numeric labels out of the language. We move that they be completely abandoned.

5. The *go to* Problem

A recent article by Irons and Feurzeig [4] proposes a method to implement recursive procedures and blocks. A minor part of their overall approach is the "Go To Interpreter," which is designed to handle a **go to** leading out of a block. Closer analysis shows, however, that the suggested Go To Interpreter must be modified to handle recursive procedures correctly.

Briefly, the method described by Irons and Feurzeig assumes that the coding for any given block will be located sequentially in memory. The beginning address '*blockbottom*' and ending address '*blocktop*' of the innermost block currently being executed are kept in a last-in, first-out list as the object program is running. When **go to** comes along, if the address of the designational expression lies within the limits of the current interval from *blockbottom* to *blocktop*, the Go To Interpreter simply jumps to that part of the program; otherwise it exits from the innermost block (doing the necessary things like releasing the array storage declared there) and recycles, comparing the address against the newly popped-up block limits.

As a simple example of where that method fails, consider:

procedure $A(L)$; **label** L;
begin $\cdots$; $A(K)$; $\cdots$;
K: **go to** L; $\cdots$
end A.

In this procedure the dots '$\cdots$' stand for statements by which we mean to calculate something; and in case of trouble we want to exit to statement L. The procedure might call itself recursively, and in that case if trouble occurs it wants to exit to K, which will then exit all the way out to L. A little study will convince the reader that a procedure of this kind would not be unnatural in practice. Yet if the Go To Interpreter would try to go to K, it would not exit the procedure as it should; in fact it would get into a tight loop.

The following solution to this **go to** problem should always work, and it also has the advantage of not requiring the program to be stored sequentially.

a) The value of a designational expression has three parts: a machine address (to which we wish to jump eventually), the block number in which this address is local, and the level of recursion to which this block is currently nested (the number of times this block has been entered minus the number of times it has been left).

b) The Go To Interpreter is given such a value. It compares the block number and recursion number with the topmost block number and its current recursion number; if they are equal, we simply jump, otherwise we exit from the block and start the process over, considering the new block in the same way.

6. "Undefined"

Peter Naur has recently pointed out [7] that the following sequence is undefined:

```
integer I; ···
begin procedure A; I := I + 1;
   begin real I;
      A;
   end;
end A.
```

Here the procedure A uses a nonlocal variable I. There is nothing illegal about this; but then I is redeclared in another block, and A is called again. Perhaps the nonlocal variable I is now to be the redeclared I? No, it is undefined by virtue of the stipulation in ALGOL Report 4.7.6. We say that the call of A is "outside the scope of the outer I" although the word "outside" is probably misleading; it really is "inside a hole in the scope of the outer I."

Maybe it's good to leave such things undefined. But a programmer who writes such a procedure obviously wants A to maintain a global count of how often it has been invoked, and doesn't want the procedure to become undefined just because the name it has chosen for the counter happens to have been redeclared. Thus a more useful language would have defined the situation so that A always accesses the variable I in the scope of its declaration, not a different variable in the scope of its call.

Here is a more complicated example of something that is undefined in ALGOL 60. It is similar to the example we gave earlier for labels:

```
procedure A(L); switch L;
begin switch K := J; ··· ;
   A(K); ··· ;
J: go to L[1]; ···
end A.
```

Suppose we call A, then it calls itself recursively with K as the actual parameter, and we somehow get to statement J. At that point we want to go to $L[1]$, which is $K[1]$, which is J; but this J is *outside* the

scope of the J that belonged to the switch parameter K, so the result is *undefined*, according to ALGOL Report 5.3.5. (See also paragraph 4.7.3.2. As we recursively call A we rename the local switch K to, say, K', as K is a parameter whose name conflicts with switch K. But J is not a parameter, so we do not change J to J' but we in fact go out of the scope of the outer J.)

A similar example seems to reveal an inconsistency in ALGOL 60. Consider:

```
switch S := J;
procedure A(T); switch T;
begin ··· ;
J: go to T[1]; ···
end A;
···
J: A(S);
```

We claim that this example is undefined according to the ALGOL Report, although nothing tricky has been used. For when we call $A(S)$, we can insert the identifier S into the procedure body without conflict; but we cannot legitimately call $S[1]$ outside the scope of the label J that appears in the declaration of S. This reasoning leads to a curious restriction: When using a procedure that has switches as parameters, you must make sure that none of the labels you can go to through those switches have the same identifier as a label that is local to the procedure body.

In paragraph 4.3.5 of the Report, a **go to** statement is not supposed to transfer control if the designational expression is a switch designator whose value is undefined. We sincerely hope that this "undefined" is different from the "undefined" of paragraph 5.3.5! Furthermore, just what does "undefined" mean in 4.3.5? Does it mean simply that the subscript expression is not between 1 and the number of items in the switch, or does it mean that *any* of the subscripts are out of bounds as we proceed to evaluate the designational expressions (possibly evaluating many more switches before we come to a label)? Or perhaps one of the subscript expressions is $0 \uparrow 0$ or some other undefined value?

The "**go to** an undefined switch designator" seems in any event an impossible thing to implement efficiently on a machine. The object program will have to make unnecessary tests to determine whether or not something is defined, and this effort seems wasted because the use of an undefined switch does not seem like a particularly powerful tool in the first place.