THE EARLY DEVELOPMENT OF PROGRAMMING LANGUAGES

by

Donald E. Knuth
Luis Trabb Pardo

STAN-CS-76-562
AUGUST 1976

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

The Early Development of Programming Languages

by Donald E. Knuth and Luis Trabb Pardo

Computer Science Department
Stanford University
Stanford, California 94305

Abstract.

This paper surveys the evolution of "high level" programming languages during the first decade of computer programming activity. We discuss the contributions of Zuse ("Plankalkül", 1945), Goldstine/von Neumann ("Flow Diagrams", 1946), Curry ("Composition", 1948), Mauchly et al. ("Short Code", 1950), Burks ("Intermediate PL", 1950), Rutishauser (1951), Böhm (1951), Glennie ("AUTOCODE", 1952), Hopper et al. ("A-2", 1953), Laning/Zierler (1953), Backus et al. ("FORTRAN", 1954-1957), Brooker ("Mark I Autocode", 1954), Kamynin/Liubimskii ("ПП-2", 1954), Ershov ("ПП", 1955), Grems/Porter ("BACAIC", 1955), Elsworth et al. ("Kompiler 2", 1955), Blum ("ADES", 1956), Perlis et al. ("IT", 1956), Katz et al. ("MATH-MATIC", 1956-1958), Hopper et al. ("FLOW-MATIC", 1956-1958), Bauer/Samelson (1956-1958). The principal features of each contribution are illustrated; and for purposes of comparison, a particular fixed algorithm has been encoded (as far as possible) in each of the languages. This research is based primarily on unpublished source materials, and the authors hope that they have been able to compile a fairly complete picture of the early developments in this area.

This article was commissioned by the Encyclopedia of Computer Science and Technology, ed. by Jack Belzer, Albert G. Holzman, and Allen Kent, and it is scheduled to appear in vol. 6 or vol. 7 of that encyclopedia during 1977.

The preparation of this paper has been supported in part by National Science Foundation grant MCS 72-03752 A03, by the Office of Naval Research contract N00014-76-C-0330, and by IBM Corporation. Reproduction in whole or in part is permitted for any purpose of the United States Government.

The Early Development of Programming Languages

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 which has long been undervalued by computer scientists. After learning a high-level language, a person often tends to think mostly of improvements 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. The real depth of this subject can only be properly perceived when we realize how long it took to develop the important concepts which we now regard as self evident. These 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, and it follows that 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[*] -- will be expressed as a program in each language we discuss. Informal explanations of this program should then suffice to capture the essence of the corresponding language, although the TPK algorithm will of course not exhaust that language's capabilities; once we have understood the TPK program, we will be able to discuss the most important language features 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 author of the corresponding language; and 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 which we shall be using so frequently can be written as follows in a dialect of Algol 60.

```
TPK: begin integer i; real y; real array a[0:10];
        real procedure f(t); real t; value t;
           f := sqrt(abs(t)) + 5 × t ↑ 3;
        for i := 0 step 1 until 10 do read(a[i]);
        for i := 10 step -1 until 0 do
        begin y := f(a[i]);
           if y > 400 then write(i,"TOO LARGE")
                      else write(i,y);
        end
     end.
```

[*] Cf. "Grimm's Law" in comparative linguistics, and/or the word "typical", and/or the names of the authors of this article.

(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 above program means: Line _1_ says that i is an integer-valued variable, while y takes on floating-point approximations to real values; and $a_0, a_1, \ldots, a_{10}$ are also real valued. Lines _2_ and _3_ define the function $f(t) = \sqrt{|t|} + 5t^3$, for use in the algorithm proper which starts on line 4. Line _4_ reads in the values $a_0, a_1, \ldots, a_{10}$, in this order; then line _5_ says to do lines 6, 7, 8, 9 (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. (If a truly useful algorithm were being considered here, it would need to be much longer in order to illustrate as many different programming language features.)

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 incapable of handling the TPK algorithm exactly as presented above; so 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 $\sqrt{x}$. Secondly, if the language does not provide for alphabetic output, the string "TOO LARGE" will be replaced by the number 999 . Thirdly, some languages do not provide for input and output at all; in such a case, we shall assume that the input values $a_0, a_1, \ldots, a_{10}$ have somehow been supplied by an external process, and that our job is to compute 22 output values $b_0, b_1, \ldots, b_{21}$. Here $b_0, b_2, \ldots, b_{20}$ will be the respective " i values" 10, 9, ..., 0 , and the alternate positions $b_1, b_3, \ldots, b_{21}$ will contain the corresponding $f(a_i)$ values and/or 999 codes. Finally, if a language does not allow the programmer to define his own functions, the statement " y := f(a[i]) " will essentially be replaced by its expanded-out form " y := sqrt(abs(a[i])) + 5 × a[i] ↑ 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 the 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, p. 295] for a paraphrase of Euclid's presentation of "Euclid's algorithm". The description of algorithms was always informal, however, rendered in natural language.

During the ensuing centuries, mathematicians never did invent a good notation for dynamic processes, 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. Thus: (a) The three-address code for Babbage's machine was 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, pp. 68, 286-297]. (b) In 1914, Leonardo Torres y Quevedo used natural language to describe the steps of a short program for his hypothetical automaton; and Helmut Schreyer gave an analogous description in 1939 for the machine he had helped Konrad Zuse to build [see RA 73, pp. 95-98, 167]. (c) An example MARK I program given in 1946 by Howard Aiken and Grace Hopper [see RA 73, pp. 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 July 9, 1836 that machines as well as people could produce programs as output:

> 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. [RA 73, p. 349]

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 giving what amounts to macro-expansions 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 sends the machine to state $f(\underset{\sim}{C},\underset{\sim}{B},\underset{\sim}{a})$; it will immediately begin to scan left (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 $\underset{\sim}{C}$ while still scanning the a , and in the second case it enters state $\underset{\sim}{B}$ 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 were 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 computations involved would be expressed by means of equations. There was no concept of assignment (i.e., 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 fantastic ["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 which 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, p. 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, p. 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 [p. 26]:

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

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

> Unstarre Rechenpläne constitute the true discipline of higher combinatorial computing; however, they cannot yet be treated in this place. [ZU 44, p. 31]

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 which 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 ever written before. Among other things, there were algorithms for sorting; 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 no chess boards or people to play chess with [ZU 72, pp. 32, 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 provide 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.

```
1    A2 = (A9, AΔ1)
2    P1     |R(V)  ⇒  R
3          V| 0       0
4          A| Δ1      Δ1

5           | √|V| + 5 × V³  ⇒  R
6          V| 0           0     0
7          A| Δ1          Δ1    Δ1

8    P2     | R(V)  ⇒  R
9          V|   0      0
10         A| 11×Δ1  11×2

11          | W2(11) ┌ R1(V)  ⇒  Z
12         V|        │ 0  0      0
13         K|        │    i
14         A|        │    Δ1     Δ1

15          |        │ Z > 400   →̣   (i,+∞)  ⇒  R ┌(10-i)
16         V|        │ 0                        0 │
17         K|        │                           ┘
18         A|        │ Δ1              9         2        9
                       _______
19          |        │ Z > 400   →̣   (i,Z)   ⇒  R ┌(10-i)
20         V|        │ 0               0        0 │
21         K|        │                           ┘
22         A|        └ Δ1              9  Δ1    2        9
```

Line 1 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 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; and this idea could be repeated, in order to obtain arrays of any desired dimension. Furthermore m could be "$\square$", meaning a list of variable length, and Zuse made good use of such list structures in his algorithms dealing with graphs, algebraic formulas, and chessplay.

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 1950's, in IBM's Commercial Translator. The idea eventually appeared in many other languages, such as FACT, COBOL, PL/I, and extensions of ALGOL 60; cf. [CL 61] and [SA 69, p. 325].

Integer variables in the Plankalkül were represented by type A9. Another special type was used for floating-binary numbers, namely

$$A\Delta 1 = (3 \times S0, 7 \times S0, 22 \times S0) \quad .$$

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

(LO , LOOO , LOOLOOOOOOOOOOOOOOOOOO) .

[The +'s and -'s notation has its bits numbered 0,1,... from left-to-right, while the L's and O's notation corresponds to the more familiar binary notation, putting most significant bits at the left.] There was a special representation for "infinite" and "very small" and "undefined" quantities; for example,

$$+\infty = (LL0, L0000, 0) \ .$$

Note that the above program uses $+\infty$ instead of 999 on line <u>15</u>, since such a value seems an appropriate way to render the concept "TOO LARGE" .

Let us return now to the program itself. Line <u>1</u> introduces the data type A2 , namely an ordered pair whose first component is an integer (type A9) and whose second component is floating-point (type AΔ1). This data type will be used later for the 11 outputs of the TPK algorithm. Lines <u>2</u> thru <u>7</u> define the function f(t) , and lines <u>8</u> thru <u>22</u> 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 <u>11</u> thru <u>14</u> must be read as a unit. The second line of each group (labelled " V") is used to identify the subscripts for quantities named on the top line; thus $\begin{matrix} R, & V, & Z \\ 0 & 0 & 0 \end{matrix}$ 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, $\begin{matrix} V \\ 0 \\ i \end{matrix}$ 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 zig-zag bar from the K-line up to the top line, as in line <u>17</u> of the above 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 <u>18</u> of our example means that R_0 is of type A2 ; the " Δ1 " means that Z_0 is floating-point (type AΔ1); and the " 9" means that i is an integer. Thus each " A" in the left margin is implicitly attached to all types in its line.

Zuse remarked [ZU 45, p. 10] that the number of possible data types was so large, 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 this information is usually redundant. This is probably one of the main reasons he introduced the peculiar multi-line format. Incidentally, a somewhat similar multi-line notation has been used in recent years to describe musical notes [SM 73]; it is interesting to speculate if this notation will evolve in the same way that programming languages have.

We are now ready to penetrate further into the meaning of the above code. Each plan begins with a specification part ["Randauszug"], stating the types of all inputs and outputs. Thus, lines 2 thru 4 mean that P1 is a procedure that takes an input V_0 of type $A\Delta 1$ (floating point) and produces R_0 of the same type. Lines 8 thru 10 say that P2 maps V_0 of type $11 \times A\Delta 1$ (namely, a vector of 1 floating-point numbers, the array a_i of our TPK algorithm) into a result R_0 of type $11 \times A2$ (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 5 thru 7 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, p. 15] that $\underset{3}{Z} + 1 \Rightarrow \underset{3}{Z}$ was analogous to to the more traditional equation $\underset{3.i}{Z} + 1 = \underset{3.i+1}{Z}$. (Incidentally, the publishers of [ZU 48] used the sign $\succeq$ instead of $\Rightarrow$, but Zuse never actually wrote $\succeq$ himself.) Note 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 thru 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 $\underset{0}{R1}(x)$ on line 11 stands for the result R_0 of applying procedure P1 to input x. Lines 15 thru 18 of the program mean " if $Z_0 > 400$ then $R_0[10-i] := (i, +\infty)$ "; note Zuse's new notation $\underset{\bullet}{\rightarrow}$ for conditionals. Lines 19 thru 22 are similar, the bar over " $Z_0 > 400$ " indicating 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 [cf. ZU 72, p. 28; ZU 45, p. 32]; this idea has recently been revived in the BLISS language [WR 71].

The reader should now be able to understand the above code completely. In the text accompanying his programs in Plankalkül notation, Zuse made it a point to state also the mathematical relations between the variables which appeared. He called such a relation an impliciter Ansatz; we would now call it an "invariant". This was yet another fundamental idea about programming; and, like Zuse's data structures, it disappeared from programming languages during the 1950's, waiting to be enthusiastically received when the time was ripe [HO 71].

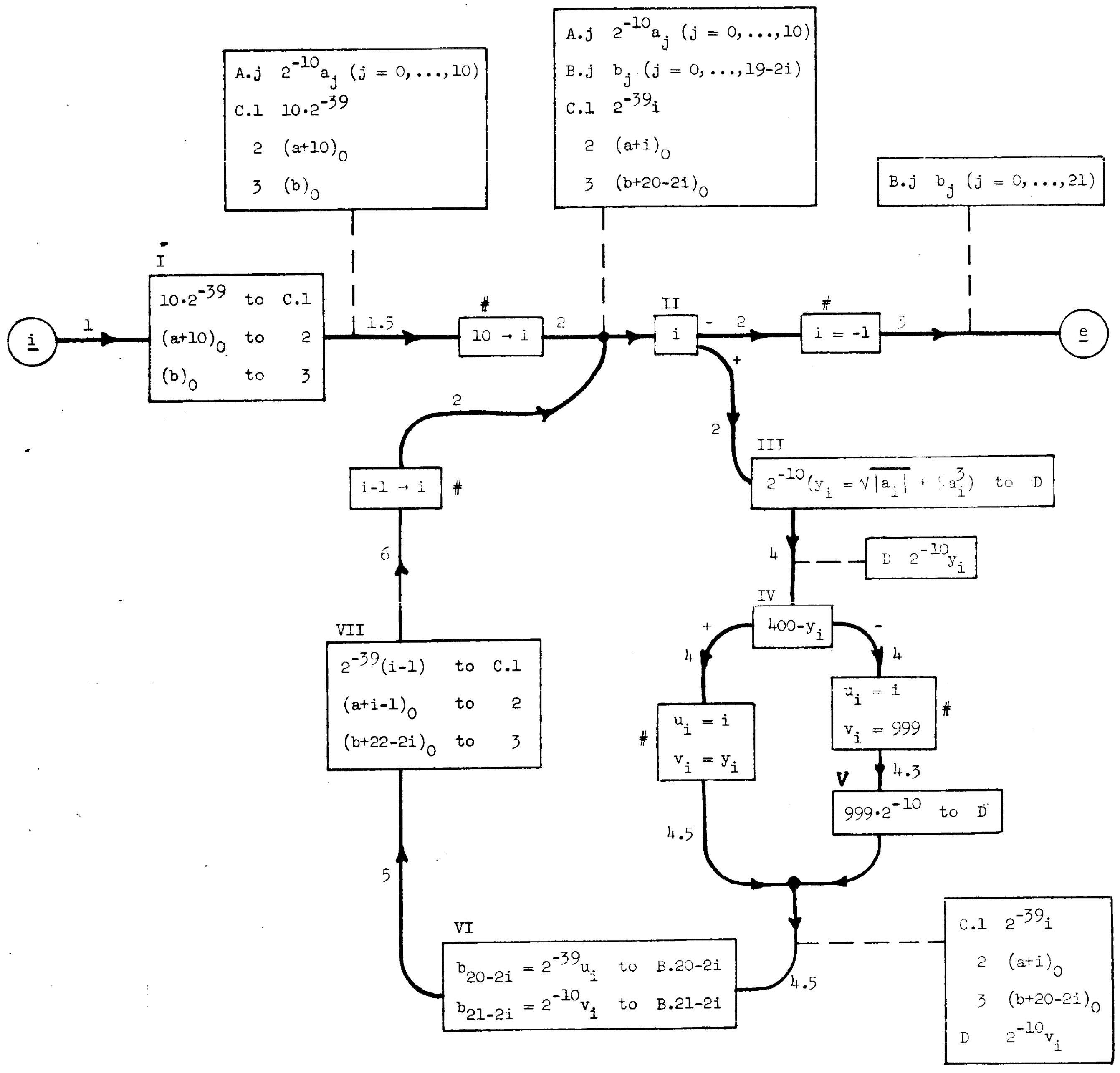
Zuse had visions of using the Plankalkül some day as the basis of a programming language that could be translated by machine (cf. [ZU 72, pp. 5, 18, 33, 34]); but in 1945, he was considering first things first -- namely, he needed to decide what concepts should be embodied in a notation for programming. We can summarize his accomplishments by saying that the Plankalkül incorporated 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, p. 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; and 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, pp. 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 this work to that of Zuse. There are striking differences, such as an emphasis on numerical calculation rather than on data structures; and there are also 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 which has entered our language as both noun and verb.

We all know what flowcharts are; but comparatively few people 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. Here is how the TPK algorithm would probably have looked, if Goldstine and Von Neumann had been asked to deal with it in 1947:


A.j $2^{-10}a_j$ ($j = 0,\dots,10$)
C.1 $10\cdot2^{-39}$
2 $(a+10)_0$
3 $(b)_0$
A.j $2^{-10}a_j$ ($j = 0,\dots,10$)
B.j b_j ($j = 0,\dots,19-2i$)
C.1 $2^{-39}i$
2 $(a+i)_0$
3 $(b+20-2i)_0$
B.j b_j ($j = 0,\dots,21$)
i
1
I
$10\cdot2^{-39}$ to C.1
$(a+10)_0$ to 2
$(b)_0$ to 3
1.5
#
$10 \to i$
2
II
i
-
2
#
$i = -1$
3
e
+
2
2
III
$2^{-10}(y_i = \sqrt{|a_i|} + 5a_i^3)$ to D
#
$i-1 \to i$
6
4
D $2^{-10}y_i$
IV
$400-y_i$
+
-
4
4
VII
$2^{-39}(i-1)$ to C.1
$(a+i-1)_0$ to 2
$(b+22-2i)_0$ to 3
#
$u_i = i$
$v_i = y_i$
$u_i = i$
$v_i = 999$
#
4.3
V
$999\cdot2^{-10}$ to D
4.5
5
VI
$b_{20-2i} = 2^{-39}u_i$ to B.20-2i
$b_{21-2i} = 2^{-10}v_i$ to B.21-2i
4.5
C.1 $2^{-39}i$
2 $(a+i)_0$
3 $(b+20-2i)_0$
D $2^{-10}v_i$

]]>

Several things need to be explained about this original notation, and probably the most important consideration is the fact that the boxes containing " $10 \to i$ " and " $i-1 \to i$ " were not intended to specify any computation. This amounts to a significantly different viewpoint than we are now accustomed to, and the reader will find it worthwhile to ponder this conceptual difference until he or she understands it. The box " $i-1 \to i$ " 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 " $i-1 \to i$ " and go thru 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 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 . (Cf. [GV 47, §§ 7.6, 7.7].)

There were four kinds of boxes in a flow diagram: (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 which says " $i = -1$ ", and two which assert 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 which helped to prove the validity of the program and which might help the programmer to write code for the operation boxes.

The next most prominent feature about original flow-diagrams is the fact that a programmer was required to be conscious of the scaling (i.e., the binary point location) of all numbers in the computer memory. A computer word was 40 bits long and its contents was to be regarded as a binary fraction x in the range $-1 \leq x < 1$. Thus, for example, the above flowchart assumes that $2^{-10}a_j$ is initially present in storage position A.j, rather than the value a_j itself; and the outputs b_j are similarly scaled.

The final mystery which 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; and 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 a programmer would use his problem-solving ability, together with his 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 artificial intelligence, and it was far beyond the state of the art in 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 it 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] which 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:

> 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. [CU 50, ¶ 34]

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:

$$\begin{aligned}
F(t) &= \{\sqrt{|t|} + 5t^3 : A\} \\
I &= \{10:i\} \to \{t = L(a+i)\} \to F(t) \to \{A:y\} \\
&\qquad \to II \to It_7(0,i) \to O_1 \,\&\, I_2 \\
II &= \{x = L(b+20-2i)\} \to \{i:x\} \to III \\
&\qquad \to \{w = L(b+21-2i)\} \to \{y:w\} \\
III &= \{y > 400\} \to \{999:y\} \,\&\, O_1
\end{aligned}$$

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 and substitute it into all appearances of 'x' in the following instruction groups".

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

I_j denotes the j-th entrance of this routine, namely the beginning of its j-th instruction group.

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

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

$It_7(m,i) \to O_1 \,\&\, O_2$ means "decrease i by 1 , then if $i \geq m$ go to O_2 , otherwise to O_1 ".

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']:

> 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 [i.e., subroutines] to make the simpler sorts of programs -- these would be programmed directly -- but only to avoid repetitions in programs of some complexity. Nevertheless, there are three reasons for pushing clear back to formation of the simplest programs from the basic programs [i.e., 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 he would have constructed for $F(t)$, if t^3 were replaced by $t\cdot t\cdot t$, is

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

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. Late 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 of 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, they could not imagine wasting any extra computer time for something a programmer could do by himself. Mauchly had greater foresight in this regard; and J. R. Logan put it this way:

> 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. [RR 55]

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 = WO , t = TO , y = YO .

Eleven inputs go respectively into words UO , T9 , T8 , ... , TO .

```
Constants:   ZO = 000000000000
             Z1 = 010000000051   [1.0  in floating-decimal form]
             Z2 = 010000000052   [10.0]
             Z3 = 040000000053   [400.0]
             Z4 = ΔΔΔTOOΔLARGE
             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  WO  03  Z2
01   0:  y = (√ abs t) + 5 cube t               TO  02  07  Z5  11  TO
02                                              00  YO  03  09  20  06
03       y 400    if≤to 1                       00  00  00  YO  Z3  41
04       i print, 'TOO LARGE' print-and-return  00  00  Z4  59  WO  58
05       0  0     if=to 2                       00  00  00  ZO  ZO  72
06   1:  i print, y print-and-return            00  00  YO  59  WO  58
07   2:  TO UO shift                            00  00  00  TO  UO  99
08       i = i-1                                00  WO  03  WO  01  Z1
09       0  i   if≤to 0                         00  00  00  ZO  WO  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 value	1n	(n+2)nd power	59	print and return carriage
02	(	07	+	2n	(n+2)nd root	7n	if= to n
03	=	08	pause	4n	if≤to n	99	cyclic shift of memory
04	/	09	)	58	print and tab	Sn, Tn, ..., Zn	quantities

Thus, " i = 10 " would actually be coded as the word " 00 00 00 W0 03 Z2 " as shown; packets of 00 's 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 which continuously 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 last. This fact needed to be understood by the programmer, who had to break long equations up appropriately into several words (cf. 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 above program means " temp = T0, T0 = T1, ..., T9 = U0, U0 = temp "; and fortunately this facility is all that the TPK algorithm needs.

The following press release from Remington Rand appeared in Journal of the ACM, 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, p. 243] (calling it "short-order code"), [HO 53, p. 142] ("short-code"), [HO 58, p. 165] ("Short Code"). In [HM 53, p. 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 a 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.

> 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. [BU 51, p. 12]

In 1950, Burks sketched a so-called "Intermediate Programming Language" which was to be the step one notch above the Internal Program Language. Instead of spelling out complete rules for this Intermediate Programming 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:

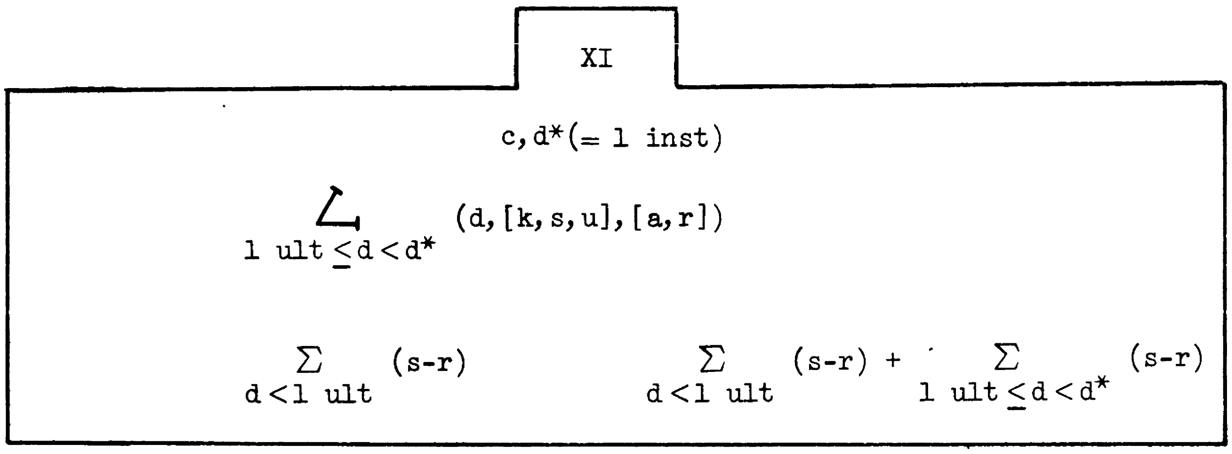
```
    1.   $10 \rightarrow i$
 To 10.


 From  1,35
      10.   $A+i \rightarrow 11$                Compute location of  $a_i$
      11.   $[A+i] \rightarrow t$               Look up  $a_i$  and transfer to storage
      12.   $|t|^{1/2}+5t^3 \rightarrow y$       $y_i = \sqrt{|a_i|} + 5a_i^3$
      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.   $999 \rightarrow y$                 $v_i = 999$
 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 indicate Burks's style of writing comments at that time; and they succeed in making the program almost completely self-explanatory. Note that the assignment operation is well established by now; and Burks used it also in the somewhat unusual form " $i \rightarrow i{+}1$ " shown in the comment to instruction 34 [BU 50, p. 41].

The prime symbol which appears within instruction 30 meant that the computer was to save this intermediate result, as it was a common subexpression that could be used later without recomputation. Burks mentioned that several of the ideas embodied in this language were due to Janet Wahr, Don Warren, and Jesse Wright.

> 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. [BU 51, p. 13]

At the other end of the spectrum, nearer to Ordinary Business Language, Burks and his colleagues later proposed an abstract form of description which may be of independent interest, even though it does not relate to the rest of our story. The following example suffices to give the flavor of their "first Abstraction Language", proposed in 1954:


XI
c, d*(= 1 inst)
$\angle_{1\ \mathrm{ult} \le d < d^*} (d, [k, s, u], [a, r])$
$\sum_{d < 1\ \mathrm{ult}} (s-r)$
$\sum_{d < 1\ \mathrm{ult}} (s-r) + \sum_{1\ \mathrm{ult} \le d < d^*} (s-r)$

FORM XI: CUSTOMER'S STATEMENT

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 , in order by the first component x_1 ; and the meaning of the second line is "a listing, in order of 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 ; [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 the reasons explained by Grace Hopper in [HO 58, p. 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 (E.T.H.) in Zürich; 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:

> 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). [RU 63]

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 out 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:

```
1    Für i = 10(-1)0
2    $a_i \Rightarrow t$
3    (Sqrt Abs t) + (5 x t x t x t) $\Rightarrow$ y
4    Max(Sgn(y-400), 0) $\Rightarrow$ h
5    Z $0_i \Rightarrow b_{20-2i}$
6    (h x 999) + ((1-h) x y) $\Rightarrow b_{21-2i}$
7    Ende Index i
8    Schluss
```

Since no "if ... then" construction --much less go to -- was present in his language, the computation of

$$\begin{cases} y\ , & \text{if}\ \ y \le 400\ , \\ 999\ , & \text{if}\ \ y > 400\ , \end{cases}$$

has been done here in terms of the Max and Sgn functions he did have, plus appropriate arithmetic; see lines 4 and 6. (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 (i.e., subscripts) were completely tied to Für - Ende loops. The above program therefore invokes a trick to get i into the main formula on line 4; " $Z\ 0_i$ " is intended to use the Z instruction which transfered an indexed address to the accumulator in Rutishauser's machine [RU 52, p. 10], and it is possible to write this 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 later wrote a paper 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 $a_i = 100 + i$, $b_i = 200 + i$, $0 = 300$, $1 = 301$, $5 = 302$, $400 = 303$, $999 = 304$, $y = 305$, $h = 306$, $t = 307$:

```
    Für   i = 10  (-1)  0
1   10^12 , 50 , 10 ,  -1  , 0 , Q ,

    begin stmt    a    sub i     ⇸       t
2     010000   , 100 , .001  , 200000 , 307 , Q ,

    begin stmt     (       t     Abs     dummy    Sqrt
3     010000   , 010000 , 307 , 110000 ,   0    , 350800 ,

    dummy     )          +          (         5        x        t        x
      0    , 2000000 , 020000 , 010000 , 302 , 060000 , 307 , 060000 ,

     t       x        t       )         ⇸        y
    307 , 060000 , 307 , 200000 , 200000 , 305 , Q ,

    begin stmt     (        (        y       -        400       )          Sgn
4     010000   , 010000 , 010000 , 305 , 030000 , 303 , 200000 , 100000 ,

    dummy     )          Max       0        ⇸        h
      0    , 200000 , 080000 , 300 , 2000000 , 306 , Q ,

    begin stmt            Z       0    sub i       ⇸        b_20    sub -2i
5     010000   , 0 , 230000 , 0 ,  .001 , 200000 , 220 ,  -.002  , Q ,

    begin stmt     (        h       x       999      )          +         (
6     0100000  , 010000 , 306 , 060000 , 304 , 200000 , 020000 , 010000 ,

      (        1       -        h        )         x        y        )         ⇸
    010000 , 301 , 030000 , 306 , 200000 , 060000 , 305 , 200000 , 200000 ,

    b_21   sub -2i
    221 ,  -.002  , Q ,

    Ende
7   Q , Q ,

    Schluss
8     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 $\Rrightarrow$ (which has the same code as right parenthesis). Subscripted variables whose address is $\alpha + \sum c_j i_j$ 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 above source input. 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.

Machine instruction		Symbolic form
230010	200050	$10 \rightarrow \mathrm{Op}$, $\mathrm{Op} \rightarrow i$,
230001	120000	$1 \rightarrow \mathrm{Op}$, $-\mathrm{Op} \rightarrow \mathrm{Op}$,
200051	230000	$\mathrm{Op} \rightarrow i'$, $0 \rightarrow \mathrm{Op}$
200052	220009	$\mathrm{Op} \rightarrow i''$, $*+1 \rightarrow \mathrm{IR}_9$
239001	200081	$1+\mathrm{IR}_9 \rightarrow \mathrm{Op}$, $\mathrm{Op} \rightarrow L_1$
000000	230100	No-op , loc a $\rightarrow$ Op
200099	010050	$\mathrm{Op} \rightarrow T$, $i \rightarrow \mathrm{Op}$
020099	210001	$\mathrm{Op}+T \rightarrow \mathrm{Op}$, $\mathrm{Op} \rightarrow \mathrm{IR}_1$
011000	200307	$a_i \rightarrow \mathrm{Op}$, $\mathrm{Op} \rightarrow t$
010307	110000	$t \rightarrow \mathrm{Op}$, $\lvert\mathrm{Op}\rvert \rightarrow \mathrm{Op}$
220009	350800	$*+1 \rightarrow \mathrm{IR}_9$, go to Sqrt
000000	000000	no-op, no-op
200999	010302	$\mathrm{Op} \rightarrow P_1$, $5 \rightarrow \mathrm{Op}$

Machine Instruction		Symbolic form
060307	060307	$Op \times t \rightarrow Op$, $Op \times t \rightarrow Op$
060307	200998	$Op \times t \rightarrow Op$, $Op \rightarrow P_2$
010999	020998	$P_1 \rightarrow Op$, $Op+P_2 \rightarrow Op$
200305	010305	$Op \rightarrow y$, $y \rightarrow Op$
030303	200999	$Op-400 \rightarrow Op$, $Op \rightarrow P_1$
010999	100000	$P_1 \rightarrow Op$, Sgn $Op \rightarrow Op$
200998	010998	$Op \rightarrow P_2$, $P_2 \rightarrow Op$
080300	200306	$\mathrm{Max}(Op,0) \rightarrow Op$, $Op \rightarrow h$,
230000	200099	$0 \rightarrow Op$, $Op \rightarrow T$
010050	020099	$i \rightarrow Op$, $Op+T \rightarrow Op$
210001	230220	$Op \rightarrow IR_1$, loc $b_{20} \rightarrow Op$
200099	230002	$Op \rightarrow T$, $2 \rightarrow Op$
120000	060050	$-Op \rightarrow Op$, $Op \times i \rightarrow Op$
020099	210002	$Op+T \rightarrow Op$, $Op \rightarrow IR_2$
010000	231000	$(0) \rightarrow Op$, $IR_1 \rightarrow Op$
202000	230221	$Op \rightarrow b_{20-2i}$, loc $b_{21} \rightarrow Op$
200099	230002	$Op \rightarrow T$, $2 \rightarrow Op$
120000	060050	$-Op \rightarrow Op$, $Op \times i \rightarrow Op$
020099	210001	$Op+T \rightarrow Op$, $Op \rightarrow IR_1$
010301	030306	$1 \rightarrow Op$, $Op-h \rightarrow Op$
200999	010306	$Op \rightarrow P_1$, $h \rightarrow Op$
060304	200998	$Op \times 999 \rightarrow Op$, $Op \rightarrow P_2$
010999	060305	$P_1 \rightarrow Op$, $Op \times y \rightarrow Op$
200997	010998	$Op \rightarrow P_3$, $P_2 \rightarrow Op$
020997	201000	$Op+P_3 \rightarrow Op$, $Op \rightarrow b_{21-2i}$
010081	210009	$L_1 \rightarrow Op$, $Op \rightarrow IR_9$
010050	220008	$i \rightarrow Op$, $*+1 \rightarrow IR_8$
030052	388003	$Op-i'' \rightarrow Op$, to (IR_8+3) if $Op = 0$
010050	020051	$i \rightarrow Op$, $Op+i' \rightarrow Op$
200050	359000	$Op \rightarrow i$, to (IR_9)
000000	999999	no-op , stop
999999		stop

(Several bugs on pp. 39-40 of [RU 52] needed to be corrected in order to produce this code, but Rutishauser's original intent was 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 above code has the interesting property that it is completely relocatable -- even if we move all instructions up or down by one-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 <u>every</u> 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	$\pi' \to A$
	base address 100 for	$\underline{100} \to i$
	the input array a).	$B \to \pi$
B.	Let a new input a_i be	$\pi' \to B$
	given. Increase i by unity,	$? \to \downarrow i$
	and proceed to C if $i > 10$,	$i+\underline{1} \to i$
	otherwise repeat B .	$[(\underline{1} \cap (i \dot{-} \underline{110}))\cdot C]+[(\underline{1} \dot{-} (i \dot{-} \underline{110}))\cdot B] \to \pi$
C.	Set $i = 10$.	$\pi' \to C$
		$\underline{110} \to i$
D.	Call x the number a_i ,	$\pi' \to D$
	and prepare to calculate	$\downarrow i \to x$
	its square root r (using	$E \to X$
	subroutine R), returning	$R \to \pi$
	to E .	
E.	Calculate $f(a_i)$ and	$\pi' \to E$
	attribute it to y .	$r+\underline{5}\cdot\downarrow i\cdot\downarrow i\cdot\downarrow i \to y$
	If $y > 400$, continue	$[(\underline{1} \cap (y \dot{-} \underline{400}))\cdot F]+[(\underline{1} \dot{-} (y \dot{-} \underline{400}))\cdot G] \to \pi$
	at F, otherwise at G.	
F.	Output the actual value	$\pi' \to F$
	of i , then the value	$i \dot{-} \underline{100} \to ?$
	999 ("too large").	$\underline{999} \to ?$
	Proceed to H.	$H \to \pi$

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

Here comments in an approximation to Böhm's style appear 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 ", i.e., 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 this type of statement to set the initial value of variable 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"; 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 $x \mathbin{\dot{-}} y$ was the logician's subtraction operator,

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

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

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

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

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

$$\begin{aligned} C \to \pi \;, &\quad \text{if } \; i > \underline{110} \;; \\ B \to \pi \;, &\quad \text{if } \; i \le \underline{110} \;. \end{aligned}$$

It is easy to read Böhm's program with these notational conventions in mind. Note 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 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 $t = 2^{46}$.	$\pi' \to R$
	$\underline{0} \to r$
	$\underline{70368744177664} \to t$
	$S \to \pi$
S. If $r{+}t \le x$, go to T ,	$\pi' \to S$
otherwise go to U .	$r{+}t \dot{-} x \to u$
	$[(\underline{1}\dot{-}u)\cdot T]+[(\underline{1} \cap u)\cdot U] \to \pi$
T. Decrease x by $r{+}t$,	$\pi' \to T$
divide r by 2 , increase	$x \dot{-} r \dot{-} t \to x$
r by t , and go to V .	$r:2+t \to r$
	$V \to \pi$
U. Divide r by 2 .	$\pi' \to U$
	$r:2 \to r$
	$V \to \pi$

V. Divide t by 4 . If $t = 0$, exit, otherwise return to S .

$$\pi' \rightarrow U$$
$$t:\underline{4} \rightarrow t$$
$$[(\underline{1} \dot{-} t)\cdot X]+[(\underline{1} \cap t)\cdot S] \rightarrow \pi$$

(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, pp. 143-159]. To prove its validity, one can verify that the following invariant relations hold when we reach step S:

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

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 , the division operator " : " taking 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 which were merely labels ($\pi' \rightarrow \ldots$) . This 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, which we have seen illustrated above, was to find the leftmost pair of parentheses which have the highest level, so that they enclose a parenthesis-free formula α , and to compile the code for " $\alpha \rightarrow P_q$ "; then the subformula " (α) " 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 (cf. [KN 68, p. 457]). Here is a brief indication of how Böhm's algorithm would have translated the statement $((a:(b\cdot c))+((d\cap e)\dot{-}f)) \rightarrow g$, assuming that the bug referred to above had been removed:

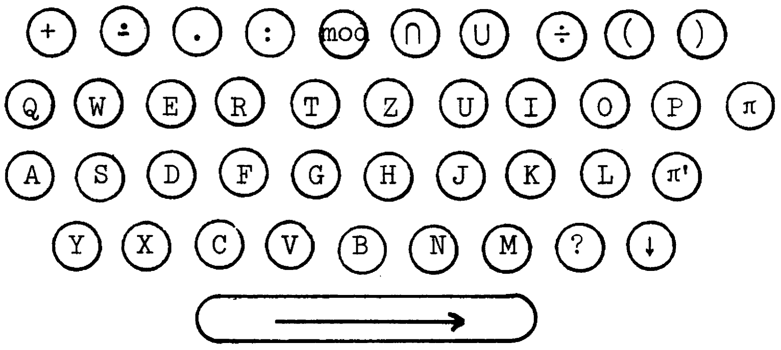
Input	Current partial instruction	Current position in tree	Contents of tree (instructions and stack pointers) ①	②	③	④	⑤
(		①	⓪				
(		②	⓪	①			
a	a	②	⓪	①			
:	a:	②	⓪	①			
(		③	⓪	a:③,①	②		
b	b	③	⓪	a:③,①	②		
•	b•	③	⓪	a:③,①	②		
c	b•c	③	⓪	a:③,①	②		
)		②	⓪	a:③,①	b•c→③		
)		①	⓪	a:③ →②	b•c→③		
+	②+	①	⓪	a:③ →②	b•c→③		
(		④	②+④,⓪	a:③ →②	b•c→③	①	
(		⑤	②+④,⓪	a:③ →②	b•c→③	①	④
d	d	⑤	②+④,⓪	a:③ →②	b•c→③	①	④
∩	d∩	⑤	②+④,⓪	a:③ →②	b•c→③	①	④
e	d∩e	⑤	②+④,⓪	a:③ →②	b•c→③	①	④
)		④	②+④,⓪	a:③ →②	b•c→③	①	d∩e→⑤
∸	⑤∸	④	②+④,⓪	a:③ →②	b•c→③	①	d∩e→⑤
f	⑤∸f	④	②+④,⓪	a:③ →②	b•c→③	①	d∩e→⑤
)		①	②+④,⓪	a:③ →②	b•c→③	⑤∸f→④	d∩e→⑤
)		⓪	②+④→①	a:③ →②	b•c→③	⑤∸f→④	d∩e→⑤
→	①	⓪	②+④→①	a:③ →②	b•c→③	⑤∸f→④	d∩e→⑤

At this point the contents of the tree would be punched out, in reverse preorder:

d∩e → ⑤
⑤∸f → ④
b•c → ③
a:③ → ②
②+④ → ①

and the following symbol " g " would evoke the final instruction " ① → 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 following keyboard [BO 52', Fig. 9]:



Constants in the source program were to be assigned a variable name and input separately.

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

Meanwhile, in England.

Our story so far has introduced us to many firsts, such as the first algebraic interpreter, the first algorithms for parsing and code generation, the first compiler in its own language. 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 from the published literature that Glennie introduced this system. When Christopher Strachey referred favorably to it in [ST 52, pp. 46-47], he did not mention Glennie's name, and it was inappropriate for Glennie to single out his own contributions when he co-authored an article with J. M. Bennett at the time [BG 53, pp. 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 1950's. 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 this way, at the beginning of a lecture he delivered at Cambridge University in February, 1953:

> 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. ... <u>To make it easy, one must make coding comprehensible</u>. 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. [GL 52]

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, p. 176]; it is described in [HA 52, pp. 36-38, 229-263, illustrated on pp. 20, 37, 230]. By pushing appropriate buttons on the console of this machine, one or more appropriate machine codes would be punched on tape for the equivalent of three-address instructions such as " -b3 x |ci| → ai " or " 1/√x9 → 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 fifty-six authors of the Harvard report [HA 52]; his visit to America in 1950 is one of the reasons he and Böhm did not get together.)

Our TPK algorithm can be expressed in Glennie's AUTOCODE as follows:

```
1    c@VA t@IC x@½C y@RC z@NC
2    INTEGERS +5 → c
3          → t
4      +t      TESTA Z
5      -t
6              ENTRY Z
7    SUBROUTINE 6 → z
8     +tt → y → x
9     +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 code!) Lines 1 - 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 1 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 2 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 3 assigns it to variable t . Line 4 means " go to label Z if t is positive "; line 5 puts -t in the accumulator; and line 6 defines label Z`. Thus the net effect of lines 4 thru 6 is to put $|t|$ into the lower accumulator. Line 7 applies subroutine 6 (integer square root) to this value, and stores it in z . On line 8 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 9 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 thru 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, up to and including 0 ; and in our case the loop was performed for n = 20,18,16,...,2,0 . These values are twice what might be expected, because the Mark I addresses were for half-words. Lines 14 thru 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 thru 31 comprise 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 in 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 thru 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 8 and 9 the upper accumulator was considered relevant (in the $\rightarrow y$), 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 ", while " +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 $\rightarrow$ was substituted for " .

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 he made at the time:

> 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. [GL 52]

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, p. 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:

> It seems advisable to concentrate less on the ability to write, say
>
> + a + b + ab → c
>
> as it is relatively easy for the programmer to write
>
> ```
> A a
> A b
> H a
> V b
> T c  .
> ```
> [MG 53, p. 79]

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

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, there was time 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, pp. 27-32]; and Wilkes later [WI 52, WI 53] extended the concept further to include general symbolic addresses (i.e., not simply relative to a small number of origins). For many years these were called "floating addresses". Similar developments in assembly systems occurred in America and elsewhere; cf. [RO 52]. (b) An artificial machine language or pseudo-code 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 these instructions, emulating the hypothetical computer. The first interpretive routines appeared in programming's first textbook, by Wilkes, Wheeler, and Gill [WW 51, pp. 34-37, 74-77, 162-164]; the primary aim of this book was to present a library of subroutines and the methodology of their use. Shortly afterwards a refined interpretive routine for floating-point calculation was described by Brooker and Wheeler [BW 53], including the ability for subroutines nested to any depth. Interpretive routines in their more familiar compact form were introduced by J. M. Bennett (cf. [WW 51, Preface and pp. 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 pseudo-codes need not be interpreted, they could also be expanded out into direct machine language instructions. She and her associates at UNIVAC proceeded to construct an experimental program which would do such a translation, and they called it a compiling routine.

> 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. [MO 54, p. 15]

(See also [HO 55, p. 22].) The first "compiler" in this sense, named A-0, was in operation in the spring of 1952, when Dr. Hopper spoke on the subject at the first ACM National Conference [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 macro expansion.

The A-0 "compiler" was improved to A-1 (January, 1953) and then to A-2 (August, 1953); the original implementors were Richard K. Ridgeway and Margaret H. Harper. Quite a few references to A-2 have appeared in the literature of those days [HM 53, HO 53, HO 53', MO 54, WA 54], but these authors 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 will see, the language was quite primitive by comparison with those we have been studying, and this is why we choose to credit Glennie with the first compiler although A-0 was completed first; yet it is important to understand what was called a "compiler" in 1954, in order to 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 to 34	36	38	40	42 - 58
10	5	400	-1	∞	4	3	a_0 to a_{10}	i	y, y', y''	t, t', t''	temp storage

Program

0.	GMI000	000002	Read input and necessary constants from T_2
	ITEM01	WS.000	
	SERVO2	BLOCKA	
	1RG000	000000	
1.	GMM000	000001	
	000180	020216	10.0 = i
	1RG000	001000	
2.	AM0034	034040	$a_{10}^2 = t$
3.	RNA040	010040	$\sqrt[4]{t} = t'$
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.	OWNΔCO	DEΔ003	
	K00000	K00000	
	F00912	E001RG	if $t'' \geq 0$, go on to Op. 10
	000000	Q001CN	
	1RG000	008040	
	1CN000	000010	
9.	GMM000	000001	
	000188	020238	'ΔΔΔTOO ΔLARGE ΔΔΔΔΔΔ ΔΔΔΔΔΔ' = y''
	1RG000	009000	
10.	YTO036	038000	Print i, y''
11.	GMM000	000001	
	000194	200220	Move 20 words from WS14 to WS40
	1RG000	011000	

```
12.  GMM000  000001
     000222  200196   Move 20 words from WS40 to WS16
     1RG000  012000
-----------------------------------------------------------
13.  ALL012  F000T1̸
     1RG000  013036   Replace i by i+(-1) and go to Op. 2
     2RG000  000037   if  i ≠ -1 , otherwise go to Op. 14
     3RG000  000006
     4RG000  000007
     5RG000  000006
     6RG000  000007
     1CN000  000002
     2CN000  000014
     1RS000  000036
     2RS000  000037
-----------------------------------------------------------
14.  OWNΔCO  DEΔ002
     810000  820000   Rewind tapes 1 and 2, and halt.
     900000  900000
     1RG000  014000
-----------------------------------------------------------
     R̸ØENDΔ  INFO.R̸
```

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 thru 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. Note that 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 in operation 13.

By 1955, A-2 had become more streamlined, and the necessity for OWN CODE in the above program had disappeared; see [PR 55] for a description of A-2 coding, vintage 1955. (Another paper [TH 55] also appeared at that time, presenting the same example program.) Operations 7 and the following of the above program could now be replaced by

```
7.   QT0038  004000   To Op. 9 if y″ > 400
     1CN000  000009
8.   QU0038  038000   Go to Op. 10
     1CN000  000010
9.   MV0008  001038  ⎫
10.  YT0036  038000  ⎪
11.  MV0014  010040  ⎪
12.  MV0040  010016  ⎪
13.  AAL036  006006  ⎬  Same meaning as before, but new syntax.
     1CN000  000002  ⎪
     2CN000  000014  ⎪
14.  RWS120  000000  ⎪
     END△CO  DING△△  ⎭
```

Laning and Zierler.

Grace Hopper was particularly active as a spokesperson for automatic programming during the 1950's; 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, p. 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 M.I.T. 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:

> 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. [HO 56]

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:

> 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. [BH 54]

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 will 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 some day 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:

```
1          v|N =    ⟨input⟩,
2          i =0,
3      1   j =i+1,
4          a|i =v|j,
5          i =j,
6          e =i-10.5,
7          CP 1,
8          i =10,
9      2   y = F^1(F^11(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 which punched paper tape and had a fairly large character set (including both upper and lower case letters); at M.I.T. they also had superscript digits $^{0}, ^{1}, \ldots, ^{9}$ and a vertical line $|$. The language used the vertical line to indicate subscripts; thus the " $5(a|i)^3$ " on line 9 means $5a_i^3$.

A programmer would insert his eleven input values for the TPK algorithm into the place shown on line 1; then they would be converted to binary notation and stored on the magnetic drum as variables $v_1, v_2, \ldots, v_{11}$. If the numbers had a simple arithmetic pattern, an abbreviation could also be used; e.g.,

```
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 1, allowing the operator to substitute a data tape at that point before reading in the rest of the source program.

Lines 2 thru 7 are a loop which 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 2 thru 7 is line 7; " CP k, " means "if the last expression computed was negative, go to the instruction labeled k".

In line 9, 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

$$Dx = y + 1,$$
$$Dy = -x,$$

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.

> 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. [AL 54, p. 64]

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

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

> 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 [M.I.T. Instrumentation] Lab had managed and operated the M.I.T. 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 2-1/2 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. [LA 76]

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

Laning describes his original parsing technique as follows:

> 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. [LA 76]

Plans for a much expanded Whirlwind compiler were dropped when the M.I.T. 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 c. 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 later 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 [1976], MAC and FORTRAN have about equal use at CSDL," according to [LA 76]; here CSDL means C. S. Draper Laboratory, the successor to M.I.T. 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 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 M.I.T. in order to see the Laning/Zierler system in operation. The big problem facing them was to implement such a language with suitable efficiency.

> 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 sceptics: ... 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. [BH 64]

By November of 1954, Backus's group had specified "The IBM Mathematical FORmula TRANslating system, <u>FORTRAN</u>". (Almost all the languages we shall discuss from now on had acronyms.) The first paragraph of their report [IB 54] emphasizes 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 places specific emphasis on the IBM 704; machine independence was not a primary goal, although a concise mathematical notation "which 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."

> 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. [IB 54]

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

Perhaps the reader thinks he knows FORTRAN already; it is certainly the earliest high-level language that is still in use. However, few people have seen the original 1954 version of FORTRAN, so it is instructive to study TPK as it might have been expressed in "FORTRAN 0":

```
1        DIMENSION A(11)
2        READ A
3     2  DO 3,8,11 J=1,11
4     3  I=11-J
5        Y=SQRT(ABS(A(I+1)))+5*A(I+1)**3
6        IF (400. >=Y) 8,4
7     4  PRINT I,999.
8        GO TO 2
9     8  PRINT I,Y
10   11  STOP
```

The READ and PRINT statements do not mention any FORMATs, although an extension to format specification was contemplated [p. 26]; programmer-defined functions were also under consideration [p. 27]. The DO statement in line 3 means, "Do statements 3 thru 8 and then go to 11"; the abbreviation " DO 8 J=1,11 " was also allowed at that time, but the original general form is shown here for fun. Note that the IF statement was originally only a two-way branch (line 6); the relation could be = , > , or >= . On line 5 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 being the first language in which a variable's name could be larger than one letter, contrary to established mathematical conventions. Note that mixed mode arithmetic was allowed, the compiler was going to convert " 5 " to " 5.0 " in line 5. A final curiosity about this program is the GO TO statement on line 8; this 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) subscripted subscripts to one level, such as A(M(I,J),N(K,L)) were allowed; (b) subscripts of the form N*I+J were allowed, provided that at least two of the variables N , I , J were declared to be "relatively constant" (i.e., 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),...,A(n)) ← (A(3),...,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 towards the implementation; however, the full job took 2.5 more years (18 man-years), 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 [BR 56, pp. 155-157] was almost the same. Brooker's emphasis on economy of description 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 Mark I Autocode are called v1, v2, ... and the integer variables -- which may be used also as indices (subscripts) -- are called n1, n2, ... The Autocode for TPK is easily readable with only a few auxiliary comments, given the memory assignments $a_i = v_{1+i}$, $y = v_{12}$, $i = n_2$:

```
1        n1 = 1              sets $n_1 = 1$
        vn1 = I              reads input into $v_{n_1}$
         n1 = n1+1
      j1,11 ≥ n1             jumps to 1 if $n_1 \leq 11$
         n1 = 11
2      * n2 = n1-1           prints $i = n_1-1$
        v12 = vn1
    j3,v12 ≥ 0·0
        v12 = 0·0-v12        sets $v_{12} = |v_{12}|$
3       v12 = F1(v12)        $(v_{12} = \sqrt{|a_i|})$
        v13 = 5·0 ⊗ vn1
        v13 = vn1 ⊗ v13
        v13 = vn1 ⊗ v13      $(v_{13} = 5a_i^3)$
        v12 = v12 + v13      $(y = f(a_i))$
   j4,v12 > 400·0
      * v12 = v12            prints y
         j5
4     * v12 = 999·0          prints 999
5        n1 = n1-1
     j2,n1 > 0               tests for last cycle
           H                 halt
          (j1)               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 " (j1) " jumps to statement 1.

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:

> 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. [BR 58, p. 16]

Gary 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 [Programmiru͡ioshchye Programmy] -- or ПП for short. An experimental program ПП-1 for the STRELA computer was constructed by E. Z. Li͡ubimskiĭ and S. S. Kamynin during the summer of 1954; and these two authors, together with M. R. Shura-Bura, E. L. Lukhovitskai͡a, and V. S. Shtarkman, completed a production compiler called ПП-2 in February, 1955. This 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. Poderi͡ugin and V. M. Kurochkin; this 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-3 completed in 1957 (see [ER 58']), the language was based on a notation for expressing programs developed by A. A. Li͡apunov in 1953. Li͡apunov'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 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 thru 14 ; and $\overset{n}{\ulcorner}$, $\underset{m}{\llcorner}$ mean respectively " go to operator n if true, go to operator m if false ", while $\overset{i}{\urcorner}$, $\underset{i}{\lrcorner}$ 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 in lieu of a flowchart. The details of operators would be written separately and input to ПП-2 after dividing them into operators of types R (relational), A (arithmetic), Z (dispatch), F (address modification), O (restoration), and N (nonstandard, i.e., machine language). In the above case, 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.	$c_6 = v_6$	[$10 = i$, i.e., 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.	$(\surd\ v_2)+(c_5 \cdot v_1 \cdot v_1 \cdot v_1) = v_3$	$[(\surd\ x)+(5 \cdot a_i \cdot a_i \cdot a_i) = y]$
A_8.	$v_6 = v_4,\ c_2 = v_5$	$[i = b_i,\ 999 = c_i]$
A_{10}.	$v_6 = v_4,\ v_3 = v_5$	$[i = b_i,\ y = c_i]$
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. $v_1, v_1, -1;\ v_4,\ v_5, +2$

[when i changes, v_1 goes down by 1, v_4 thru v_5 go up 2]

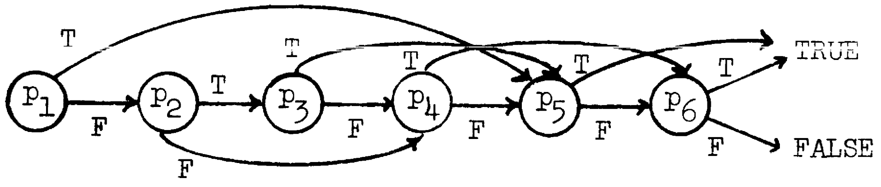
c_1.	$.1 \cdot 10^1$	[1]
c_2.	$.999 \cdot 10^3$	[999]
c_3.	0	
c_4.	$.4 \cdot 10^3$	[400]
c_5.	$.5 \cdot 10^1$	[5]
Working cells:	100,119	[compiled program can use locations 100-119 for temp storage]
v_1.	130	[initial address of a_i]
v_2.	131	[address of x]
v_3.	132	[address of y]
v_4.	133	[initial address of b_i]
v_5.	134	[initial address of c_i]
v_6.	154	[address of i]

Operator 1 initializes i , then operators 2 thru 13 are the loop on i . Operator 2 moves a_i to a fixed cell, and makes sure that operators 3 thru 6 use this fixed cell; this programmer-supplied optimization means that fewer addresses in instructions have to be modified when i changes. Operators 3 thru 5 set $x = |a_i|$, and operator 6 sets $y = f(a_i)$. (Note the parentheses in operator 6; precedence was not recognized.) Operators 7 thru 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 ΠΠ-2 are quite interesting from the standpoint of compiler history; for example, they avoided the recomputation of common subexpressions within a single formula. 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



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

1 Massiv a (11 ĭacheek) [declares an array of 11 cells]

2 $a_0 = 0$ [address in array a]

3 $a_j = -1 \cdot j + 10$ [address in array a depending on j]

4 j: $j_{nach} = 0$, $j_{kon} = 11$ [information on loop indexes]

5 0, 11, 10, 5, y, 400, 999, i [list of remaining constants and variables]

6 $(\mathrm{Ma}, 080, 0, a_0); (\mathrm{Mb}, 0, 0\bar{1}, 0);$

7 $[_j$ $10-j \Rightarrow i$; $\sqrt{}$ mod $a_j + 5 \times a_j^3 \Rightarrow y$;

8 $R(y, 0102; \overset{0101}{\ulcorner} (400, \infty));$

9 $\underset{0101}{\llcorner}$ Vyd i, ⇒ 0; Vyd 999, ⇒ 0; $\overset{0103}{\ulcorner}$;

10 $\underset{0102}{\llcorner}$ Vyd i, ⇒ 0; Vyd y, ⇒ 0; $\underset{0103}{\llcorner}$]; STOP

After declarations on lines 1 thru 5, the program appears here on lines 6 thru 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 3; note that a_j does not mean the j-th element of a , it means an element of a which depends on j . The commands in line 6 are BESM machine language instructions which read 11 words into memory starting at a_0 . Line 7 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 4.) Line 8 is a relational operator which means, "If y is in the interval $(400, \infty)$, i.e., if $y > 400$, go to label 0101 ; otherwise go to 0102 ." Labels were given as hexadecimal numbers, and the notation $\underset{n}{\llcorner}$ indicates the program location of label n . The " Vyd " instruction in lines 9 and 10 means convert to decimal, and " , ⇒ 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 above program would actually have been converted into BESM words in the form required by ПП. (The hexadecimal digits were written $0,1,\ldots,9,\bar{0},\bar{1},\ldots,\bar{5}$. 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 ПП 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 "
$0\bar{4}$	$000\ 0000\ 0000\ 02\bar{1}5$	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

Location	Contents	Meaning
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	$[_j$
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 $\surd$ mod a_j
35	2^{0}, 03 18 09 13	+ 5 x 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	$\overset{0101}{\ulcorner}$ (400,∞))
39	018 0101 0000 0000	$\underset{0101}{\llcorner}$
$3\bar{0}$	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	$\overset{0103}{\ulcorner}$
$3\bar{3}$	018 0102 0000 0000	$\underset{0102}{\llcorner}$
$3\bar{4}$	2^{0}, $\bar{5}4$ $1\bar{1}$ 07 00	Vyd i ,$\Rightarrow$ 0
$3\bar{5}$	2^{0}, $\bar{\bar{5}}4$ $1\bar{2}$ 07 00	Vyd y ,$\Rightarrow$ 0
40	018 0103 0000 0000	$\underset{0103}{\llcorner}$
41	$01\bar{5}$ $13\bar{5}\bar{5}$ $13\bar{5}\bar{5}$ $13\bar{5}\bar{5}$	]
42	$01\bar{5}$ 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 $\times$ 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. Detailed specifications of all its algorithms were published in [ER 58]; Ershov was aware of Rutishauser's work [p. 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 BACAIC, 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 X + 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, " . " for multiplication; variables are given single-letter names (except K), and constants are denoted by K1 thru K99 . The above program is to be used with the following input data:

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

Data values 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 → I ", so the initial value I = 11 needs to be input only in Case 1.

Expressions 2, 3, 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 <u>expression</u> 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 above program is an unconditional jump; expression 9 says that the value of I and expression 5 should be 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 of 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. This system is somewhat similar in flavor to $\pi\pi$-2, except that it is based on flow diagrams instead of operator schemata. They characterized its status in the following way:

> 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. [EK 55]

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

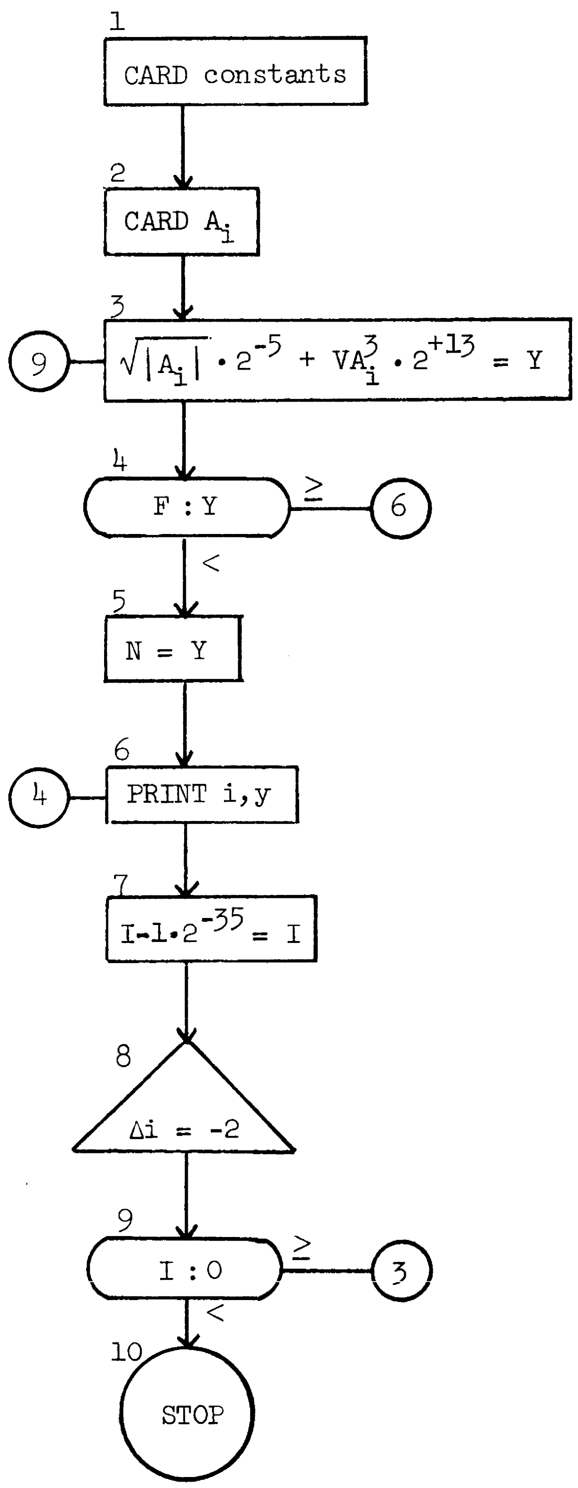
To solve the TPK problem, let us first agree to scale the numbers by writing

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

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:



Read values of constants and 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 answer.

Decrease i by 1.

Decrease address of A_i by 2 wherever it appears.

Return to 3 if $i \geq 0$.

Stop the machine.

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

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

(Addresses in the IBM 701 go by half words, but variables in KOMPILER 2 occupy full words. Address 61 denotes halfwords 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 so-called "flow diagram cards", one card per box in the flow diagram, and the "algebraic cards", one per complex equation. In our 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 14
 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 the algebraic cards are

```
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 , i.e., immediately. (The scale code ddbb means to take the 10-digit

data as a decimal fraction, multiply 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 $(.00\ldots01010)_2 = 10\cdot2^{-35}$, 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 (cf. step 3) by 2.
9. If the contents of location 61 is nonnegative, go to step 3.
10. Stop the machine.

The first two algebraic cards in the above 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

$$\sqrt{|A_i|}\cdot2^{-5}+VA_i^3\cdot2^{13} = Y \quad .$$

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 which used three-line format analogous to that of MAC (but apparently 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. This language ADES (Automatic Digital Encoding System) was presented at the ACM national meetings in 1955 [when no proceedings were published] and 1956 [BL 56"], and at the ONR symposium in 1956 [BL 56'].

> 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', p. 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, p. iii]

A full description of the language was given in a lengthy report [BL 56]; it is rather difficult to understand several aspects of ADES, 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].)

$\underline{1}$ $a_0 11 : q_0 11,$

$\underline{2}$ $\quad f_{50} = + \surd \text{ abs } c_1 \cdot \cdot \cdot 5 \; c_1 \; c_1 \; c_1,$

$\underline{3}$ $d_{12} b_1 = r_0,$

$\underline{4}$ $d_{22} b_2 = \leq b_3 \; 400, \; b_3, \; 999,$

$\underline{5}$ $\quad b_3 = f_{50} \; a_0 \; r_0,$

$\underline{6}$ $\quad r_0 = -10 \; q_0,$

$\underline{7}$ $\forall \; 0 \; q_0 \; 10 \; b_0 = f_0 \; b_1 \; b_2,$

Here is a rough translation: Line $\underline{1}$ is the so-called "computer table", meaning that input array a_0 has 11 positions, and the "independent index symbol" q_0 takes 11 values. Line $\underline{2}$ defines the auxiliary function f_{50} , our $f(t)$; arithmetic expressions were defined in Łukasiewicz's parentheses-free notation, 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, cf. [HA 57].)

Line $\underline{3}$ states that the dependent variable b_1 is equal to the dependent index r_0 ; the " d_{12} " here means that this is to be output as component 1 of a pair. Line $\underline{4}$ similarly defines b_2 , which is to be component 2. This line is a "branch equation" meaning " $\utilde{\text{if}}$ $b_3 \leq 400$ $\utilde{\text{then}}$ b_3 $\utilde{\text{else}}$ 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, p. 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 statements, Blum may well have intended that f or g not be evaluated or even defined when $\varphi = 0$ or 1 , respectively.)

Line 5 says that b_3 is the result of applying f_{50} to the r_0-th element of a_0. Line 6 explains that r_0 is $10-q_0$. Finally, line 7 is a so-called "phase equation" which 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:

> 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. [ER 58', p. 275]

The transcript of a 1965 discussion of declarative vs. 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 <u>Comm. ACM</u> 9 (1966), pp. 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 translator 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 took 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 above program 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   -    -
```

Thus 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 5 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, p. 1]

Purdue received one of the first Datatron computers, manufactured by Electrodata Corporation (cf. *J. ACM* 2 (1955), p. 122, and [PE 55]); this machine was later known 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]. It is interesting to note that their 1956 paper 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 this 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, (cf. [PS 57, p. 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', p. 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 has 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 was the first really useful compiler; IT and IT's derivatives were used successfully and frequently 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, p. 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 do not have to 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 above program assumes that the input data will specify eleven values for C1 thru 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 $f(C_{I1+1})$; 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. Note the use of mixed integer and floating-point arithmetic here. 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 $Y1 \leq 400$; 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 above program would actually be punched onto cards in the following form -- using K for comma, M for minus, Q for quote, L and R for parentheses, etc.:

```
0001   READ                       F
0002   3K I1K 10K M1K OK          F
0005   Y1 Z Q 20EK ACLI1S1R Q     F
0005         S L5 X LCLI1S1R P 3RR F
0006   G3 IF 400J0 W Y1           F
0006   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 say 1 I variable, 1 Y variable, 11 C variables, 10 statements. (It was possible to "go to" statement number n, where n was the value of any integer expression, so an array of statement locations was kept in the running program.)

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 in the Purdue Compiler language [OR 58] would look like this:

```
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"+(5x(ci0p3))  f
 6  r  g8, r y0 ≤ 400.0 f                   [go to 8 if y_0 ≤ 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 ≤ i0 f                       [go to 5 if i_0 ≥ 0]
10  h  f                                    [halt]
```

Note 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.

FORTRAN Arrives.

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, p. 40]

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

> 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. [IB 56, p. 1]
>
> Object programs produced by *Fortran* will be nearly as efficient as those written by good programmers. [p. 2]

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

> 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. [RO 64]

In spite of these difficulties, it is clear that FORTRAN I was worth waiting for; it soon was accepted even more enthusiastically than its proponents had dreamed.

> 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. [BA 58, p. 246]

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:

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. [WW 57, pp. 136-137]

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 language changes that had occurred:

```
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 are

(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, pp. 25, 30-31].

(3) Formats are 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] are 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. (Cf. [BE 57], p. 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; note the vertical bar and the superscript 3 in sentence (4), indicating an extended character set that could be used with some peripherals. 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 considering 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]. This 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, pp. 317-322].

The first experimental B-0 compiler was operating in 1956 [HO 58, p. 171], and it was released to UNIVAC customers in 1958 [SA 69, p. 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 [SB 59] describing it 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 today's pocket calculators, allowing formulas consisting only of operators, parentheses, and numbers, while their highest level includes 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:

```
1   ◇   0000.00000000 ⇒ a↓11↑
2       2.27 ⇒ a↓1↑
3       3.328 ⇒ a↓2↑
        ...
12      5.28764 ⇒ a↓11↑
13      10 ⇒ i
14  44* a↓i+1↑ ⇒ t
15      √ Bt+5×t×t×t ⇒ y
16      i = □□ ⇒ i
17      y > 400 → 77*
18      y = □□□.□□□ ⇒ y
19      →  88*
20  77* 999 = □□□.□□□ ⇒ y
21  88* i-1 ⇒ i
22      i > -1 → 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 1 specifies storage allocation; it says that a is an 11-element array whose entries will contain at most 12 digits.

Lines 2 through 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 2 through 12 could be replaced by the code

```
    1 ⇒ i
33* ●●●●●● ⇒ a↓i↑
    i+1 ⇒ i
    i < 12 → 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 above program 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.

Language	Principal Author(s)	Year	Arithmetic	Implementation	Readability	Control structures	Data structures	Machine independence	Impact	First
Plankalkül	Zuse	1945	X, S, F	F	D	A	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	X	F	A	D	C	A	F	Common subexpression notation
Klammerausdrücke	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	D	Macro expander
Algebraic interpreter	Laning/ Zierler	1953	F	B	A	D	C	A	B	Constants in formulas
Autocode	Brooker	1954	X, F	A	B	D	C	A	C	Clean two-level storage
ПП-2	Kamynin/ Liubimskiĭ	1954	F	D	C	D	C	D	D	Code optimization
ПП	Ershov	1955	F	B	B	C	C	F	C	Book about a compiler
BACAIC	Grems/Porter	1955	F	A	A	D	F	A	D	Expression-oriented
KOMPILER 2	Elsworth/Kuhn	1955	S	C	C	D	C	C	F	Scaling aids
ADES	Blum	1956	X, F	D	D	B	C	A	F	Declarative language
IT	Perlis	1956	X, F	A	B	C	C	A	B	Successful compiler, change of machine
FORTRAN I	Backus	1956	X, F	A	A	C	C	A	A	I/O formats, 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

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 very 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 to use was it?

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

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

Data structures: How high-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) 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 languages until the 1960's. Our languages today probably have too many features, but the languages up to FORTRAN I had too few.

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 (cf. [BO 58], who describes the FOR TRANSIT compiler which was developed by a group of programmers at IBM under the direction of R. W. Bemer and D. Hemmes). 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 the international project attempting to define a "standard" algorithmic language. Just after the 1955 meeting in Darmstadt, a group of European computer scientists began to plan a new language (cf. [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) 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 x t↑3↓;
         if (y > 400); y := 999;
         b[20-2xi] := i;
         b[21-2xi] := y
   end;
   return;
integer (i)
end TPK
```

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, Pa.: Remington Rand Univac, 1957).

[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, D.C.: Office of Naval Research, Dept. of the Navy, 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 "co-authorship" only ten or fifteen years later!]

[BA 54] J. W. Backus, "The IBM 701 Speedcoding system," J.ACM 1 (1954), 4-6.

[BA 58] J. W. Backus, "Automatic programming: Properties and performance of FORTRAN systems I and II," Mechanisation of Thought Processes, National Physical Laboratory Symposium No. 10, 1958 (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," Proc. Int. Conf. Inf. Processing (Paris: UNESCO, 1959), 125-131.

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

[BA 76] John Backus, "Programming in America in the Nineteen Fifties -- some personal impressions," Proc. International Research Conf. on the History of Computing (Los Alamos, 1976), to appear.

[BA 76'] F. L. Bauer, letter to D. E. Knuth dated July 7, 1976; 2 pp.

[BB 57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. Mitchell 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," Proc. Western Joint Comp. Conf. (1957), 188-197.

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

[BC 54] Arthur W. Burks, Irving M. Copi, and Don W. Warren, "Languages for analysis of clerical problems," Engineering Research Institute, Informal Memorandum 5 (Ann Arbor, Mich.: Univ. of Michigan, 1954), iii + 24 pp.

[BE 57] R. W. Bemer, "The status of automatic programming for scientific problems," Proc. 4th Annual Computer Applications Symposium, 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, ed. 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, D.C.: Office of Naval Research, Dept. of the Navy, 1954), 106-113.

[BH 64] J. W. Backus and W. P. Heising, "FORTRAN," IEEE Trans. Electronic Comp. EC-13 (1964), 382-385.

[BL 56] E. K. Blum, "Automatic Digital Encoding System II (ADES II)," NAVORD Report 4209, Aeroballistic Research Report 326, U. S. Naval Ordnance Laboratory (February 8, 1956), v + 45 pp. + (2 + 1 + 7) pp. of appendices.

[BL 56'] E. K. Blum, "Automatic Digital Encoding System, II," Symposium on Advanced Programming Methods for Digital Computers, Washington, D.C. ONR Symposium Report ACR-15 (1956), 71-76.

[BL 56"] E. K. Blum, "Automatic Digital Encoding System, II (ADES II)," Proc. ACM National Conference 6 (1956), paper 29, 4 pp.

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

[BL 57'] E. K. Blum and Shane Stern, "An ADES Encoder for the IBM 650 calculator," NAVORD Report 4412, U. S. Naval Ordnance Laboratory (December 19, 1956), 15 pp.

[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], Patent application No. 13567, filed in Milan on October 1, 1952; 26 pp. + 2 tables.

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

[BO 58] B. C. Borden, "FORTRANSIT, a universal automatic coding system," Canadian Conf. for Computing and Data Proc. (Toronto: U. of Toronto Press, 1958), 349-359.

[BP 52] J. M. Bennett, D. G. Prinz, and M. L. Woods, "Interpretative sub-routines," Proc. ACM National Conference 2 (Toronto, 1952), 81-87.

[BR 55] R. A. Brooker, "An attempt to simplify coding for the Manchester electronic computer," British J. Appl. 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 1 computer," Proc. I.E.E. 103, part B, supplement (1956), 151-157.

[BR 58] R. A. Brooker, "The Autocode programs developed for the Manchester University computers," Comp. J. 1 (1958), 15-21.

[BR 58'] R. A. Brooker, "Some technical features of the Manchester Mercury AUTOCODE programme," Mechanisation of Thought Processes, National Physical Laboratory Symposium No. 10, 1958 (London: Her Majesty's Stationery Office, 1959), 201-229.

[BR 60] R. A. Brooker, "MERCURY Autocode: Principles of the Program Library," Ann. Rev. in Automatic Prog. 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," Deutsches Patentamt, Auslegeschrift 1094019 (March 30, 1957), published December, 1960; 26 cols. plus 6 Figs.

[BS 62] Friedrich Ludwig Bauer and Klaus Samelson, "Automatic computing machines and method of operation," United States Patent Office, patent 3,047,228 (July 31, 1962); 32 cols. plus 17 Figs.

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

[BU 51] Arthur W. Burks, "An intermediate program language as an aid in program synthesis," Engineering Research Institute, Report for Burroughs Adding Machine Company (Ann Arbor, Mich.: Univ. of Michigan, 1951), ii + 15 pp.

[BW 53] R. A. Brooker and D. J. Wheeler, "Floating operations on the EDSAC," Math. 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," Comm. ACM 15 (1972), 678-685.

[CH 36] Alonzo Church, "An unsolvable problem of elementary number theory," Amer. J. Math. 58 (1936), 345-363.

[CK 56] J. Chipps, M. Koschmann, S. Orgel, A. Perlis, and J. Smith, "A mathematical language compiler," Proc. ACM National Conf. 6 (1956), paper 30, 4 pp.

[CL 61] R. F. Clippinger, "FACT - A Business Compiler: Description and comparison with COBOL and Commercial Translator," Ann. Rev. in Auto. Prog. 2 (1961), 231-292.

[CU 48] Haskell B. Curry, "On the composition of programs for automatic computing," Naval Ordnance Laboratory Memorandum 9806 (Silver Spring, Md., 1949); 52 pp. [Written in July, 1948.]

[CU 50] H. B. Curry, "A program composition technique as applied to inverse interpolation," Naval Ordnance Laboratory Memorandum 10337 (Silver Spring, Md., 1950); 98 pp. + 3 figs.

[CU 50'] H. B. Curry, "The logic of program composition," Applications scientifiques de la logique mathématique, Actes du 2^e Colloque International de Logique Mathématique, 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," Univ. of California Radiation Lab., Livermore, Calif., report UCRL-4585 (November 7, 1955), 66 pp.

[ER 58] A. P. Ershov, Programmiruîoshchaîa Programma dlîa Bystrodeistvuîoshcheǐ Elektronnoǐ Schetnoǐ Mashiny (Moscow: Akad. Nauk SSSR, 1958), 116 pp. English translation, Programming Programme for the BESM Computer (London: Pergamon, 1959), v + 158 pp.

[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," Mechanisation of Thought Processes, National Physical Laboratory Symposium No. 10, 1958 (London: Her Majesty's Stationery Office, 1959), 257-278.

[FE 60] G. E. Felton, "Assembly, interpretive and conversion programs for PEGASUS," Ann. Rev. in Automatic Prog. 1 (1960), 32-57.

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

[GL 52'] A. E. Glennie, "Automatic Coding," unpublished manuscript (undated, probably 1952), 18 pp. [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 dated September 15, 1965; 6 pp.

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

[GO 56] Max Goldstein, "Computing at Los Alamos, Group T-1," Symposium on Advanced Programming Methods for Digital Computers, Washington, D.C., ONR Symposium Report ACR-15 (1956), 39-43.

[GO 57] Saul Gorn, "Standardized programming methods and universal coding," J. ACM 4 (1957), 254-273.

[GO 72] Herman H. Goldstine, The Computer from Pascal to von Neumann (Princeton, N. J.: Princeton University Press, 1972), xi + 378 pp.

[GP 56] Mandalay Grems and R. E. Porter, "A truly automatic computing system," Proc. Western Joint Computer Conf. (1956), 10-21.

[GR 58] Robert M. Graham, "Translation between algebraic coding languages," Proc. ACM National Conf. 8 (1958), paper 29, 2 pp.

[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, N.J.: The Institute for Advanced Study, 1947-1948). Volume 1, iv + 69 pp.; Volume 2, iv + 68 pp.; Volume 3, iii + 23 pp. Reprinted in von Neumann's Collected Works, ed. by A. H. Taub, Vol. 5 (London: 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, Mass.: Harvard University Press, 1952), xi + 318 pp.

[HA 57] C. L. Hamblin, "Computer languages," Australian J. Science 20, 6 (December 1957), 135-139.

[HM 53] Grace M. Hopper and John W. Mauchly, "Influence of programming techniques on the design of computers," Proc. I.R.E. 41 (1953), 1250-1254.

[HO 52] Grace Murray Hopper, "The education of a computer," Proc. ACM National Conf. 1 (Pittsburgh, 1952), 243-250.

[HO 53] Grace Murray Hopper, "The education of a computer," Symp. on Industrial Appl. of Automatic Computing Equipment (Kansas City, Mo.: 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, Washington, D.C., ONR Symposium Report ACR-15 (1956), 1-2.

[HO 57] Grace M. Hopper, "Automatic programming for business applications," Proc. 4th Annual Computer Applications Symposium, Armour Research Foundation (1957), 45-50.

[HO 58] Grace Murray Hopper, "Automatic programming: present status and future trends," *Mechanisation of Thought Processes,* National Physical Laboratory Symposium No. 10, 1958 (London: Her Majesty's Stationery Office, 1959), 155-200.

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

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

[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, I. Ziller, "Programmer's Reference Manual: The FORTRAN Automatic Coding System for the IBM 704 EDPM," Applied Science Div. and Programming Research Dept., IBM (October 15, 1956), 51 pp.

[IB 57] International Business Machine Corporation, "Programmer's Primer for FORTRAN Automatic Coding System for the IBM 704" (1957), iii + 64 pp.

[KA 57] Charles Katz, "Systems of debugging automatic coding," *Automatic Coding,* Franklin Institute monograph no. 3 (1957), 17-27.

[KL 58] S. S. Kamynin, E. Z. Li͡ubimskiĭ, and M. R. Shura-Bura, "Ob avtomatizatsii programmirovanii͡a pri pomoshchi programmirui͡oshchei 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," *Automatic Coding,* Franklin Institute monograph no. 3 (1957), 39-55.

[KN 64] Donald E. Knuth, "Backus Normal Form vs. Backus Naur Form," *Comm. ACM* 7 (1964), 735-736.

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

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

[KN 72] Donald E. Knuth, "Ancient Babylonian algorithms," Comm. ACM 15 (1972), 671-677. Errata in Comm. ACM 19 (1976), 108.

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

[LA 65] J. H. Laning, letter to D. E. Knuth dated January 13, 1965; 1 p.

[LA 76] J. H. Laning, letter to D. E. Knuth dated July 2, 1976; 11 pp.

[LE 55] N. Joachim Lehmann, "Bemerkungen zur Automatisierung der Programmfertigung für Rechenautomaten," Elektronische Rechenmaschinen und Informationsverarbeitung - Electronic Digital Computers and Information Processing, proceedings of October, 1955, conference at Darmstadt, Nachrichtentechnische Fachberichte 4 (1956), p. 143 (including discussion).

[LJ 58] A. A. Liapunov, "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," MIT Instrumentation Laboratory report R-681 (November 1970), 23 pp.

[LZ 54] J. H. Laning, Jr., and N. Zierler, "A program for translation of mathematical equations for Whirlwind I," Engineering memorandum E-364 (Mass. Inst. of Technology: Instrumentation Laboratory, January, 1954), v + 21 pp.

[MG 53] E. N. Mutch and S. Gill, "Conversion routines," Automatic Digital Computation, Proc. of a symposium held at the National Physical Laboratory on March 25, 26, 27 & 28, 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, D.C.: Office of Naval Research, Dept. of the Navy, 1954), 15-21.

[NA 54] Navy Mathematical Computing Advisory Panel, Symposium on Automatic Programming for Digital Computers (Washington, D.C.: Office of Naval Research, Dept. of the Navy, 1954), v + 152 pp.

[OR 58] Sylvia Orgel, "Purdue Compiler: General description" (W. Lafayette, Ind.: Purdue Research Foundation, 1958), iv + 33 pp.

[PE 55] A. J. Perlis, "DATATRON," transcript of lecture given August 11, 1955; in Digital Computers and Data Processors, ed. by John W. Carr and Norman R. Scott (Ann Arbor, Mich.: University of Michigan, College of Engineering, 1956), Section VII.20.1, 3 pp.

[PE 57] Richard M. Petersen, "Automatic coding at G.E.," Automatic Coding, Franklin Institute monograph no. 3 (1957), 3-16.

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

[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] Alan J. Perlis and Joseph W. Smith, "A mathematical language compiler," Automatic Coding, Franklin Institute monograph no. 3 (1957), 87-102.

[PS 57'] A. J. Perlis, J. W. Smith, and H. R. Van Zoeren, "Internal Translator (IT): A compiler for the 650," Computation Center, Carnegie Institute of Technology (March, 1957). Part I, Programmer's Guide, 47 pp. Part II, Program Analysis, 68 pp. Addenda, 12 pp. (flow charts were promised on p. 3.12). Reprinted in Applications of Logic to Advanced Digital Computer Programming (Ann Arbor, Mich.: University of Michigan, College of Engineering, 1957). This report was also available from IBM Corp. 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 changed his life.]

[PS 58] A. J. Perlis and K. Samelson, "Preliminary report, International Algebraic Language," Comm. ACM 1, 12 (December 1958), 8-22. Also "Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming," Numer. Math. 1 (1959), 41-60. Also reprinted in Ann. Rev. in Automatic Programming 1 (1960), 269-290.

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

[RO 52] N. Rochester, "Symbolic programming," I.R.E. Trans. EC-2 (1952), 10-15.

[RO 64] Saul Rosen, "Programming systems and languages, a historical survey," Proc. Spring Joint Computer Conf. (1964), 1-16.

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

[RR 55] Remington Rand UNIVAC, UNIVAC Short Code, unpublished collection of dittoed notes. Preface by A. B. Tonik, dated Oct. 25, 1955 (1 page); preface by J. R. Logan, undated but apparently from 1952 (1 page); "Preliminary Exposition" (1952?, 22 pages, where pp. 20-22 appear to be a later replacement); "Short Code Supplementary Information, Topic One" (7 pp.); Addenda # 1,2,3,4 (9 pp.).

[RU 52] Heinz Rutishauser, "Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen" [Automatic machine-code generation on program-directed computers], Mitteilungen aus dem Inst. für angew. Math. an der E.T.H. Zürich No. 3 (Basel: Birkhäuser, 1952), ii + 45 pp.

[RU 55] Heinz Rutishauser, "Some programming techniques for the ERMETH," J. ACM 2 (1955), 1-4.

[RU 55'] Heinz Rutishauser, "Massnahmen zur Vereinfachung des Programmierens (Bericht über die in fünfjähriger Programmierungsarbeit mit der Z4 gewonnenen Erfahrungen)," Elektronische Rechenmaschinen und Informationsverarbeitung - Electronic Digital Computers and Information Processing, proceedings of October, 1955, conference at Darmstadt, Nachrichtentechnische Fachberichte 4 (1956), 26-30. English summary, "Methods to simplify programming, experiences based on five years of programming work with the Z4 computer," p. 225.

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

[SA 55] Klaus Samelson, "Probleme der Programmierungstechnik," Aktuelle Probleme der Rechentechnik, Ber. über das Int. Mathematiker-Kolloquium, Dresden, 1955 (Berlin: VEB Deutcher Verlag der Wissenschaften, 1957), 61-68.

[SA 69] Jean E. Sammet, Programming Languages: History and Fundamentals (Englewood Cliffs, N.J.: Prentice-Hall, 1969), xxx+785 pp.

[SB 59] K. Samelson and F. L. Bauer, "Sequentielle Formelübersetzung," Elektronische Rechenanlagen 1 (1959), 176-182. Also "Sequential formula translation," Comm. ACM 3 (1960), 76-83, 351.

[SM 73] Leland Smith, "Editing and printing music by computer," J. Music Theory 17 (1973), 292-309.

[ST 52] C. S. Strachey, "Logical or non-mathematical programmes," Proc. ACM National Conf. 2 (Toronto, 1952), 46-49.

[TA 56] D. Tamari, review of [BO 52], Zentralblatt für Mathematik 57 (1956), 107-108.

[TA 60] Alan E. Taylor, "The FLOW-MATIC and MATH-MATIC Automatic Programming Systems," Ann. Rev. in Auto. Prog. 1 (1960), 196-206.

[TH 55] Bruno Thüring, "Die UNIVAC A-2 Compiler Methode der automatischen Programmierung," Elektronische Rechenmaschinen und Informationsverarbeitung - Electronic Digital Computers and Information Processing, proceedings of October, 1955, conference at Darmstadt, Nachrichtentechnische Fachberichte 4 (1956), 154-156. English summary, p. 226.

[TU 36] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," Proc. London Math. Soc. (2) 42 (1936), 230-265; correction in vol. 43 (1937), 544-546.

[WA 54] John Waite, "Editing generators," Symposium on Automatic Programming for Digital Computers (Washington, D.C.: Office of Naval Research, Dept. of the Navy, 1954), 22-29.

[WA 58] F. Way III, "Current developments in computer programming techniques," Proc. 5th Annual Computer Applications Symposium, Armour Research Foundation (1958), 125-132.

[WH 50] D. J. Wheeler, "Programme organization and initial orders for the EDSAC," Proc. Royal Soc. (A) 202 (1950), 573-589.

[WI 52] M. V. Wilkes, "Pure and applied programming," Proc. ACM National Conf. 2 (Toronto, 1952), 121-124.

[WI 53] M. V. Wilkes, "The use of a 'floating address' system for orders in an automatic digital computer," Proc. Cambridge Philos. Soc. 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," Comm. 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, Mass.: Addison-Wesley Press, 1951), xi + 170 pp.

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

[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 dated 1944; Chapter 1 has been published in Berichte der Gesellschaft für Mathematik und Datenverarbeitung, No. 63 (Bonn, 1972), part 1, 32 pp. English translation, No. 106 (Bonn, 1976), 7-20.

[ZU 45] K. Zuse, "Der Plankalkül," manuscript prepared in 1945. Published in Berichte der Gesellschaft für Mathematik und Datenverarbeitung, No. 63 (Bonn, 1972), part 3, 285 pp. English translation of all but pp. 176-196 in No. 106 (Bonn, 1976), 42-244.

[ZU 48] K. Zuse, "Über den allgemeinen Plankalkül als Mittel zur Formulierung schematisch kombinativer Aufgaben," Archiv der Math. 1 (1948/49), 441-449.

[ZU 59] K. Zuse, "Über den Plankalkül," Elektron. Rechenanl. 1 (1959), 68-71.

[ZU 72] Konrad Zuse, "Kommentar zum Plankalkül," in Berichte der Gesellschaft für Mathematik und Datenverarbeitung, No. 63 (Bonn, 1972), part 2, 36 pp. English translation, No. 106 (Bonn, 1976), 21-41.