Published Papers -- Robert W. Bemer (1-30)

This document covers papers 1-60, from 1951 through 1970.
For papers 61-116, from 1971 through 1979, Click here.

1. R.W.Bemer, "Machine Method: Iterative Solution of Games",
   RAND Memorandum RM-595-PR, 1951
2. R.W.Bemer, "Polynomial relaxation coefficients",
   ACM Annual Meeting, Phila., 1955 Sep
3. R.W.Bemer, "PRINT I - A proposed coding system for the IBM Type 705",
   Proc. Western Joint Computer Conf., 1956 May
4. R.W.Bemer, "Reference Manual, PRINT I system for the 705", IBM Corp., 1956 Oct
5. R.W.Bemer, "Report on history and use of automatic coding",
   Proc. GUIDE, Second Session, Appendix Z, 1-6, 1957 Feb 13-15
6. R.W.Bemer, "Why the engineer should know computer programming",
   Automatic Control, 1957 Feb
7. R.W.Bemer, "How to consider a computer", Data Control Section,
   Automatic Control Magazine, 1957 Mar, 66-69
   No earlier published article, describing commercial timesharing (which the Internet is), has ever been found.
8. R.W.Bemer, "PRINT I - an automatic coding system for the IBM 705",
   Automatic Coding, Monograph #3, Proc. Automatic Coding Symposium,
   Franklin Institute, 1957 Apr, 29-38
9. R.W.Bemer, "The status of automatic programming for scientific computation",
   Proc. 4th Annual Computer Applications Symposium,
   Armour Research Foundation, 1957 Oct 24-25, 107-117
10. R.W.Bemer, "A machine method for square root computation",
    Commun. ACM 1, No. 1, 6-7, 1958 Jan
11. R.W.Bemer, "XTRAN - a compatible superstructure to FORTRAN,with growth
    possibilities", Proc. GUIDE, 1958 May
    An ALGOL predecessor from IBM
12. R.W.Bemer, "A subroutine method for calculating logarithms",
    Commun. ACM 1, No. 5, 5-7, 1958 May
13. R.W.Bemer, "Editor's note on series approximation truncation",
    Commun. ACM 1, No. 9, 3-5, 1958 Sep
14. R.W.Bemer, "Simulation programs", Commun. ACM 1, No. 11, 5-6, 1958 Nov
15. R.W.Bemer, "A checklist of intelligence for programming systems",
    Commun. ACM 2, No. 3, 8-13, 1959 Mar
16. R.W.Bemer, "A table of symbol pairs", Commun. ACM 2, No. 4, 10-11, 1959 Apr
17. R.W.Bemer, "Evaluating intelligence for programming systems",
    Automatic Control, 1959 Apr, 22-24
18. R.W.Bemer, "A proposal for a generalized card code of 256 characters",
    Commun. ACM 2, No. 9, 19-23, 1959 Sep
    -- Computing Reviews 00025
    Early public hint of 8-bit bytes to come.
19. R.W.Bemer, W.Buchholz, "An extended character set standard",
    IBM Tech. Pub. TR00.18000.705, 1960 Jan, rev. TR00.721, 1960 Jun
    -- Computing Reviews 00813

20. R.W.Bemer, "A proposal for character code compatibility",
Commun. ACM 3, No. 2, 71-72 (1960 Feb)
-- Computing Reviews 00320
-- Computer Abstracts 60-865
Here was disclosed Bemer's invention of escape sequences. (See it)
21. R.W.Bemer, "Comment on COBOL",
Management and Business Automation, 1960 Mar 22
22. R.W.Bemer, "Do it by the numbers - digital shorthand",
Commun. ACM 3, No. 10, 530-536, 1960 Oct
-- Computer Abstracts 61-199
-- Picture and story, NY Herald Tribune, front page,
Section 2, 1960 Jul 05
**This method programmed for RCA by C. Berners-Lee,
father of Tim Berners-Lee, father of the Web.
Also treated in David Kahn's classic "The Codebreakers",
and acknowledged as probably unbreakable then.**
23. R.W.Bemer, "Survey of coded character representation",
Commun. ACM 3, No. 12, 639-641, 1960 Dec
-- Computing Reviews 00639
-- Computer Abstracts 61-287
**Exposing the Babel of internal computer codes,
the impetus for the creation of ASCII.**
24. R.W.Bemer, "Data compression system",
IBM Technical Disclosures Bull. 3, 8-9, 1961 Jan
25. R.W.Bemer, "Survey of modern programming techniques",
The Computer Bulletin 4, No. 4, 127-135, 1961 Mar
-- Computing Reviews 01213
-- Computer Abstracts 61-1025
26. R.W.Bemer, "Editor's note on binary reciprocals of decimal integers",
Commun. ACM 4, No. 4, 116, 1961 Apr
-- Computer Abstracts 61-1909
27. R.W.Bemer, H.J.Smith, Jr., F.A.Williams,
"Design of an improved transmission/data processing code",
Commun. ACM 4, No. 5, 212-217, 225, 1961 May
-- Computer Abstracts 61-1920
ASCII in its original form.
28. M.Grems, R.W.Bemer, F.A.Williams, "IBM Glossary for Information Processing", 1961
29. R.W.Bemer, "The present status, achievement, and trends of programming
for commercial data processing", chapter in Digitale Informationswandler,
Vieweg & Sons, Braunschweig, Germany, 1962, J. Wiley, 1962, 312-349
-- Computer Abstracts 62-1630
30. R.W.Bemer, "An international movement in programming languages", in
Computer Applications Symp., Spartan Books, Baltimore, MD, 1962, 204-214

SUBJECT: MACHINE METHOD - ITERATIVE SOLUTION OF GAMES

By: Robert W. Bemer

The method outlined here is for the solution of a 20 x 20 game with three-digit elements in its pay-off matrix, as programmed for the Card-Programmed Electronic Calculator. It may be used for the solution of any game where the pay-off matrix has 20 or fewer rows and columns, such as an 8 x 17 matrix. It may also be used, with a slight loss in accuracy, for matrices with elements of more than three digits; in this case, all elements should be reduced by a constant factor to make the largest corrected element equal to 999. If any elements are negative a constant must be added to all elements to make them positive. The number of iterations required for the solution of the game depends upon the number of digits of accuracy required. It is conceivable that a sixth digit may be required in the running sums if accurate work is being done. (You will note that only five storage positions have been allowed for each of the running sums.) This situation may be remedied in the middle of the process by subtracting a constant, C, from each of the running sums and then adding $\frac{C}{n}$ to each of the values of $\overline{V}_n$ and $\underline{V}_n$ from that point on. The mechanics of the iteration process are explained in the paper P-78B by Dr. George Brown of The RAND Corporation and may be further understood by examination of the illustrations attached to this paper. They are:

Fig. 1 Wiring of 417 plug-board.

Fig. 2 Program Sheet for 604 plug-board.

Fig. 3 A 20 x 20 sample pay-off matrix with unique elements.

Fig. 4 and 5 Key sheets which give the row and column numbers of any element in the sample matrix. (Note that for convenience the rows and columns are numbered 10 through 29 rather than the conventional 1 through 20.)

Fig. 6	Sheet indicating the punches in all cards required.
Fig. 7	A portion of the list actually obtained in the solution of the sample game.

It will be seen that the basic deck consists of two starter cards, twenty or fewer row cards, two row sum cards, a spacer card, twenty or fewer column cards, two column sum cards and a spacer card. This deck (with the exception of the starter cards, which are removed after the first pass) is fed into the 417 repeatedly, each time representing a line of iteration. This basic deck is usually reproduced several times to form a convenient handful. The specific functions of these cards are:

1. Starter cards - Clear the storage where the pass or line number is maintained and determine a specific starting row. This arbitrary choice is made on the 604 board by altering the digits emitted on programs 2 and 3. (See programs 1 to 4, Fig. 2).

2. Row cards - Feed in columnwise the elements of a specific row through the field selector, add these new elements individually and again columnwise to the previous row sums, determine which of these amended sums is minimum and thereby make the choice of the next column to be added in. Referring to specific programs in Fig. 2:

Programs 5 to 12 - provide for indication of the column number of the row element currently being added. This is done by emitting a 10 on the first row card and raising the number by 1 for each new element. On program 9 the starting value of R_n min is established at 10.

Programs 13 to 17 - add the element cumulatively to the old row sum.

Programs 18 to 22 - test the new row sum against the standing minimum row sum; if it is smaller only, the old minimum is replaced by the new and the columnwise indication correspondingly replaced.

Programs 23 to 24 - build up the accumulative sum of row sums, pertinent to the checking device incorporated in the board.

Programs 25 to 26 - send the various new sums to storage on Channel C.

It may be noted here that Channel B reads in ten digits each time a row or column card is fed, the left hand five carrying the row sum for a specific column indication, the right hand five carrying the column sum for a specific row indication. Thus the row sum in column 13 and the column sum in row 13 are stored side by side, but only one sum is amended at a time.

Regarding placement of the various elements on the row cards: For convenience in wiring through the field selector, columns 14 through 73 carry, in three digit divisions, the elements for rows 10, 20, 11, 21, 12, 22, 13, ..., 28, 19, 29. Columns 12, 13 carry the column indication. This same arrangement occurs in the row sum cards.

3. Row sum cards - Carry the sum of all the elements in a specific row as a five digit figure, the left-most two (thousands) digits on the 130 card and the second three (units) on the 131 card. Referring to specific programs of Fig. 2:

Programs 27 to 29 - get the full five digit row sum in the counter.

Programs 30 to 34 - check to see that the row sum plus the old sum of row sums = the old sum of row sums plus the individual elements of the row accumulated one at a time. If this is not so, an error has occurred and the program repeat hub is impulsed stopping the computation.

Programs 35 to 45 - obtain the running value of $\underline{V}_n$, to three whole numbers and two decimals rounded, for the specific line just calculated. This board may be used on a 40 program machine by substituting a single program of RO-MQ, RI-GS_2 for programs 37 to 45, giving an unrounded value of $\underline{V}_n$.

Program 46 - sends the sum of row sums to counter-group 2 for storage until the next 131 card comes up.

The second row sum card contains a 9 in the operation code to summary punch from GS_1 the next column chosen to be played, $\underline{V}_n$ from GS_2, and the line or pass number from GS_3. If it is desired that only these values shall list on the 417, turn Set-up Change Switch No. 1 off.

4. Spacer card - Allows time after summary punch. Program 47 reads the next column to be played into the counter for listing.

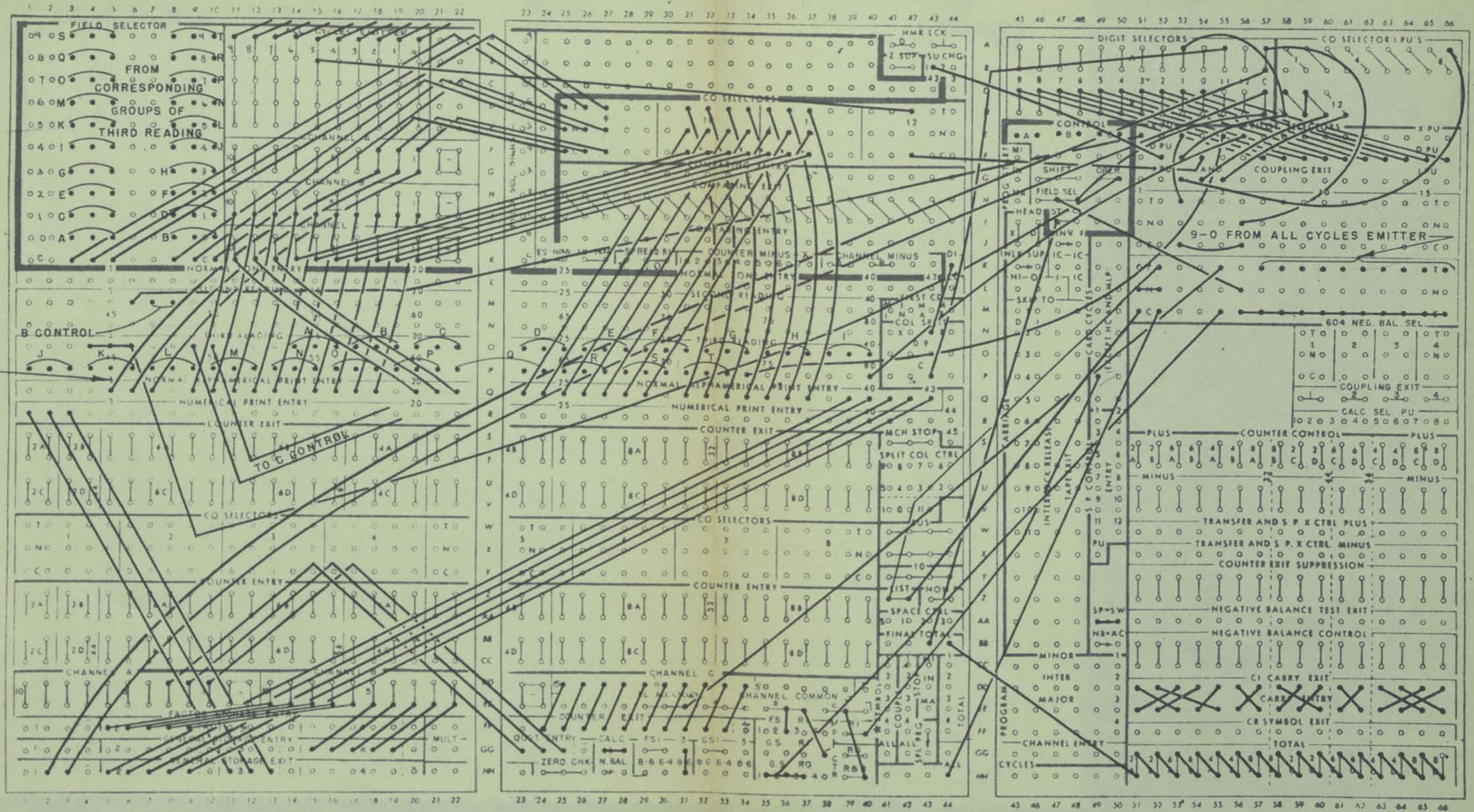
5, 6, and 7. Column cards, column sum cards and spacer card - Read the same as Items 2, 3, and 4, substituting row for column, column for row, maximum for minimum, C_n max for R_n min, larger for smaller and $\overline{V}_n$ for $\underline{V}_n$, counter group 3 for counter group 2.

In order to expedite effective duplication of this set-up, the sample matrix of Fig. 3 has been set up to show exactly how the decisions of play are made. The elements of this sample matrix consist of the numbers from 100 to 499 arranged randomly. Whenever possible an interpreted copy of the basic card deck for this matrix will be furnished along with the paper work. Also appended is Fig. 7, an actual portion of the solution of the game starting with row 11 as the initial choice. If this is duplicated it may be assumed that your boards are in working order. Within the 119 passes shown, $\underline{V}_n$ max = 303.21 (pass 109) and $\overline{V}_n$ min = 311.13 (pass 119), showing the actual value of the game to be between these limits.

IBM

CARD PROGRAMMED ELECTRONIC CALCULATOR, TYPE 402-417 CONTROL PANEL

20 x 20 MATRIX
(3 DIGITS)



FOR LISTING, RAISE ZERO-SPLITS
A- 7, 13, 18, 24, 29, 37, 43, N- 3

FIG. 1

PRINTED IN U.S.A.

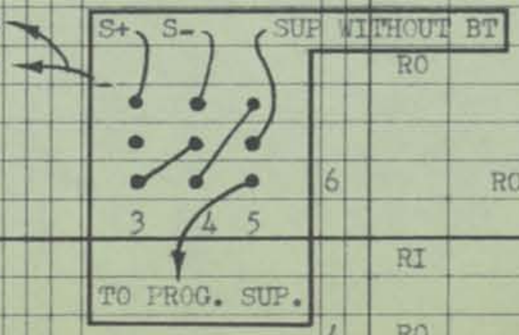
Operation 1. Starter card
Operation 2. First card of each row and column group
Operation 3. Row cards only
Operation 4. Column cards only
Operation 5. Both Row and Column cards
Operation 6. 130 card
Operation 7. 131 card
Operation 8. Spacer card

20 x 20 3D MATRIX

ELECTRONIC PROGRAM — SUPPRESSION

ENTRY FROM CHANNELS A & B — 10 POSITIONS EACH

NUMBER	OPERATION CALC-SELECTORS								GROUP SUPPRESS								BAL TEST		ZERO TEST		REPEAT	OPERATION	SHIFT		FACTOR STORAGE				MULT. QUOT.	COUNTER	GENERAL STORAGE			
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	+	—	ZERO OPEN	NON ZERO OPEN			IN	OUT	1 R^1_n C_n	2 sum of row sums	3	4	element		1 min R_n max C_n	2 min R_Σ max C_Σ	3 line	4 sum of col sums
																										xxxxx			xxx					xxxxx
1	x																																RI	
2	x																					EMIT 1 OR 2	2							RI+				
3	x																					EMIT A DIGIT								RI+				
4	x																													RO and RC	RI			
5		x																							RI		—RI—							
6					x																				RO					RI+				
7		x			x																	EMIT 1	2		(on transf. sel. #2 only)					RI+				
8					x																				RI					RO and RC				
9		x																							RO						RI			
10		x	x																											RI+			RO	
11		x	x						(thru both selectors)													EMIT 1								RI+				
12		x	x																											RO and RC			RI	
13					x																								RO	RI+				
14			x																							RO				RI+				
15				x																										RI+				RO
16			x																							RI				RO				
17				x																										RC				RI
18		x																												RO		RI		
19					x																	BALANCE TEST								RI≥		RO		
20					x																									RI+		RO		
21					x												S	S												RO		RI		
22					x												S	S							RO						RI			
23					x																						—RO—			RI+				
24					x																						—RI—			RO and RC				
25					x																		6			RO				RI+				
26					x																									RI+				RO
27						x																			RI				RO					
28							x																						RO	RI+				
29							x																4		RO					RI+				
30							x																							RI+				RO
31							x																6			RO				RI+				
32							x																				—RO—			RI-				
33							x															ZERO TEST, RC												
34							x												S		x													
35							x																3							RI+		RO		
36							x															DIVIDE							xxx.xx				RO	
37							x																							RO and RC		RI		
38							x																2							RI+		RO		
39							x																						RO			RI		
40							x															DIVIDE							x				RO	
41							x																							RO and RC				
42							x																						RO	RI+				
43							x																2							RI+		RO		
44							x															HALF ADJUST												
45							x																	2						RO and RC		RI		
46							x																				—RO—			RI+				
47								x																						RI+	RO			
48																																		
49																																		
50																																		
51																																		
52																																		
53																																		
54																																		
55																																		
56																																		
57																																		
58																																		
59																																		
60																																		



SUMMARY PUNCH
ROW OR COLUMN CHOSEN
RUNNING APPROX. TO VALUE
LINE

NUMBER — OPERATION CALC-SELECTORS 1 2 3 4 5 6 7 8 — GROUP SUPPRESS 1 2 3 4 5 6 7 8 — BAL TEST + — — ZERO TEST ZERO OPEN, NON ZERO OPEN — REPEAT — OPERATION — SHIFT IN OUT — SUPPRESSION

FACTOR STORAGE 1 2 3 4 — MULT. QUOT. — COUNTER — GENERAL STORAGE 1 2 3 4

ELECTRONIC PROGRAM

ENTRY ONLY

EXIT TO CHANNEL C – 14 POSITIONS

Indicate PICK-UP "P" SUPPRESS "S" SUPPRESS on plus "+" on minus "—" on Zero "O" Non Zero φ and Drop Out as "D"

604 PLANNING CHART

FIG. 2

SAMPLE 20 x 20 MATRIX

Col Row	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
10	222	456	275	261	234	369	204	108	100	392	289	387	487	134	220	240	227	139	176	344
11	181	476	394	292	232	142	281	159	437	225	495	484	163	365	122	114	248	458	340	153
12	199	354	249	283	161	389	287	221	212	136	499	378	198	126	235	426	433	431	102	467
13	154	443	337	113	179	200	397	399	461	125	193	407	334	364	215	118	356	226	400	105
14	210	422	130	412	352	175	485	291	483	329	442	383	325	206	213	453	252	160	311	374
15	269	294	316	107	117	414	488	157	332	290	341	115	263	247	367	103	497	112	454	464
16	486	306	375	366	257	145	430	353	379	239	211	428	444	132	258	498	301	324	371	274
17	185	233	377	101	129	201	140	186	493	445	469	144	466	463	174	333	170	172	262	285
18	278	120	477	468	460	403	450	357	189	420	169	298	438	480	370	408	312	401	345	360
19	121	409	359	427	489	391	188	385	109	277	441	180	255	246	465	195	166	415	425	326
20	173	307	310	349	223	137	271	336	300	421	254	244	123	346	362	475	424	388	299	322
21	164	384	156	416	347	474	440	376	288	241	417	124	390	490	224	196	286	207	405	250
22	317	348	320	398	165	110	462	319	177	182	106	260	457	152	279	439	146	190	133	373
23	237	435	202	131	155	168	314	191	147	361	318	455	372	273	219	303	296	158	104	396
24	323	342	492	293	217	335	268	459	429	236	192	138	216	338	282	339	331	251	363	315
25	151	419	449	479	482	284	393	128	351	382	184	231	119	245	229	167	265	148	238	343
26	406	162	321	178	313	491	270	141	410	214	472	302	218	418	452	209	295	135	111	259
27	187	432	266	470	297	253	267	434	276	327	402	280	308	116	380	413	197	264	309	143
28	355	448	473	208	149	411	183	127	496	447	368	203	395	471	451	150	330	328	256	171
29	194	228	436	242	230	381	305	478	481	358	404	386	423	243	446	205	304	272	350	494

Fig. 3

KEY-POSITION OF ELEMENTS IN SAMPLE MATRIX

Element	Row	Col	Element	Row	Col	Element	Row	Col	Element	Row	Col
100	10	18	150	28	25	200	13	15	250	21	29
101	17	13	151	25	10	201	17	15	251	24	27
102	12	28	152	22	23	202	23	12	252	14	26
103	15	25	153	11	29	203	28	21	253	27	15
104	23	28	154	13	10	204	10	16	254	20	20
105	13	29	155	23	14	205	29	25	255	19	22
106	22	20	156	21	12	206	14	23	256	28	28
107	15	13	157	15	17	207	21	27	257	16	14
108	10	17	158	23	27	208	28	13	258	16	24
109	19	18	159	11	17	209	26	25	259	26	29
110	22	15	160	14	27	210	14	10	260	22	21
111	26	28	161	12	14	211	16	20	261	10	13
112	15	27	162	26	11	212	12	18	262	17	28
113	13	13	163	11	22	213	14	24	263	15	22
114	11	25	164	21	10	214	26	19	264	27	27
115	15	21	165	22	14	215	13	24	265	25	26
116	27	23	166	19	26	216	24	22	266	27	12
117	15	14	167	25	25	217	24	14	267	27	16
118	13	25	168	23	15	218	26	22	268	24	16
119	25	22	169	18	20	219	23	24	269	15	10
120	18	11	170	17	26	220	10	24	270	26	16
121	19	10	171	28	29	221	12	17	271	20	16
122	11	24	172	17	27	222	10	10	272	29	27
123	20	22	173	20	10	223	20	14	273	23	23
124	21	21	174	17	24	224	21	24	274	16	29
125	13	19	175	14	15	225	11	19	275	10	12
126	12	23	176	10	28	226	13	27	276	27	18
127	28	17	177	22	18	227	10	26	277	19	19
128	25	17	178	26	13	228	29	11	278	18	10
129	17	14	179	13	14	229	25	24	279	22	24
130	14	12	180	19	21	230	29	14	280	27	21
131	23	13	181	11	10	231	25	21	281	11	16
132	16	23	182	22	19	232	11	14	282	24	24
133	22	28	183	28	16	233	17	11	283	12	13
134	10	23	184	25	20	234	10	14	284	25	15
135	26	27	185	17	10	235	12	24	285	17	29
136	12	19	186	17	17	236	24	19	286	21	26
137	20	15	187	27	10	237	23	10	287	12	16
138	24	21	188	19	16	238	25	28	288	21	18
139	10	27	189	18	18	239	16	19	289	10	20
140	17	16	190	22	27	240	10	25	290	15	19
141	26	17	191	23	17	241	21	19	291	14	17
142	11	15	192	24	20	242	29	13	292	11	13
143	27	29	193	13	20	243	29	23	293	24	13
144	17	21	194	29	10	244	20	21	294	15	11
145	16	15	195	19	25	245	25	23	295	26	26
146	22	26	196	21	25	246	19	23	296	23	26
147	23	18	197	27	26	247	15	23	297	27	14
148	25	27	198	12	22	248	11	26	298	18	21
149	28	14	199	12	10	249	12	12	299	20	28

Fig. 4

KEY-POSITION OF ELEMENTS IN SAMPLE MATRIX

Element	Row	Col	Element	Row	Col	Element	Row	Col	Element	Row	Col
300	20	18	350	29	28	400	13	28	450	18	16
301	16	26	351	25	18	401	18	27	451	28	24
302	26	21	352	14	14	402	27	20	452	26	24
303	23	25	353	16	17	403	18	15	453	14	25
304	29	26	354	12	11	404	29	20	454	15	28
305	29	16	355	28	10	405	21	28	455	23	21
306	16	11	356	13	26	406	26	10	456	10	11
307	20	11	357	18	17	407	13	21	457	22	22
308	27	22	358	28	19	408	18	25	458	11	27
309	27	28	359	19	12	409	19	11	459	24	17
310	20	12	360	18	29	410	26	18	460	18	14
311	14	28	361	23	19	411	28	15	461	13	18
312	18	26	362	20	24	412	14	13	462	22	16
313	26	14	363	24	28	413	27	25	463	17	23
314	23	16	364	13	23	414	15	15	464	15	29
315	24	29	365	11	23	415	19	27	465	19	24
316	15	12	366	16	13	416	21	13	466	17	22
317	22	10	367	15	24	417	21	20	467	12	29
318	23	20	368	28	20	418	26	23	468	18	13
319	22	17	369	10	15	419	25	11	469	17	20
320	22	12	370	18	24	420	18	19	470	27	13
321	26	12	371	16	28	421	20	19	471	28	23
322	20	29	372	23	22	422	14	11	472	26	20
323	24	10	373	22	29	423	29	22	473	28	12
324	16	27	374	14	29	424	20	26	474	21	15
325	14	22	375	16	12	425	19	28	475	20	25
326	19	29	376	21	17	426	12	25	476	11	11
327	27	19	377	17	12	427	19	13	477	18	12
328	28	27	378	12	21	428	16	21	478	29	17
329	14	19	379	15	18	429	24	18	479	25	13
330	28	26	380	27	24	430	16	16	480	18	23
331	24	26	381	29	15	431	12	27	481	29	18
332	15	18	382	25	19	432	27	11	482	25	14
333	17	25	383	14	21	433	12	26	483	14	18
334	13	22	384	21	11	434	27	17	484	11	21
335	24	15	385	19	17	435	23	11	485	14	16
336	20	17	386	29	21	436	29	12	486	16	10
337	13	12	387	10	21	437	11	18	487	10	22
338	24	23	388	20	27	438	18	22	488	15	16
339	24	25	389	12	15	439	22	25	489	19	14
340	11	28	390	21	22	440	21	16	490	21	23
341	15	20	391	19	15	441	19	20	491	26	15
342	24	11	392	10	19	442	14	20	492	24	12
343	25	29	393	25	16	443	13	11	493	17	18
344	10	29	394	11	12	444	16	22	494	29	29
345	18	28	395	28	22	445	17	19	495	11	20
346	20	23	396	23	29	446	29	24	496	28	18
347	21	14	397	13	16	447	29	19	497	15	26
348	22	11	398	22	13	448	28	11	498	16	25
349	20	13	399	13	17	449	25	12	499	12	20

Fig. 5

COMPONENTS OF BASIC DECK - SAMPLE MATRIX

PERMANENT PUNCHES

THESE COLUMNS CHANGE WITH A DIFFERENT MATRIX

col	col	col	col	col	columns		columns		DESCRIPTION
			1	11	111	111	667	777	
123	456	78	90	23	456	789	890	123	
	1		BLANK	CARD					STARTER CARDS
110	235	84	74	10	222	173	121	194	ROW CARDS
111	35x	11	11	11	456	307	409	228	
112	35x	12	12	12	275	310	359	436	
113	35x	13	13	13	261	349	427	242	
114	35x	14	14	14	234	223	489	230	
115	35x	15	15	15	369	137	391	381	
116	35x	16	16	15	204	271	188	305	COMMON XY IN COLUMN 74
117	35x	17	17	17	108	336	385	478	
118	35x	18	18	18	100	300	109	481	
119	35x	85	75	19	392	421	277	447	
120	35x	86	76	20	289	254	441	404	
121	35x	21	21	21	387	244	180	386	
122	35x	22	22	22	487	123	255	423	
123	35x	23	23	23	134	346	246	243	
124	35x	24	24	24	220	362	465	446	
125	35x	25	25	25	240	475	195	205	
126	35x	26	26	26	227	424	166	304	
127	35x	27	27	27	139	388	415	272	
128	35x	28	28	28	176	299	425	350	
129	35x	87	77	29	344	322	326	494	
130	66x	00	00	30	005	006	006	006	ROW SUM CARDS
131	799	82	72	31	264	064	269	860	
132	888	00	00	00					SPACER CARD
210	245	84	74	10	222	289	392	344	COLUMN CARDS
211	45x	11	11	11	181	495	225	153	
212	45x	12	12	12	199	499	136	467	
213	45x	13	13	13	154	193	125	105	
214	45x	14	14	14	210	442	329	374	
215	45x	15	15	15	269	341	290	464	COMMON XY IN COLUMN 80
216	45x	16	16	16	486	211	239	274	
217	45x	17	17	17	185	469	445	285	
218	45x	18	18	18	278	169	420	360	
219	45x	85	75	19	121	441	277	326	
220	45x	86	76	20	173	254	421	322	
221	45x	21	21	21	164	417	241	250	
222	45x	22	22	22	317	106	182	373	
223	45x	23	23	23	237	318	361	396	
224	45x	24	24	24	323	192	236	315	
225	45x	25	25	25	151	184	382	343	
226	45x	26	26	26	406	472	214	259	
227	45x	27	27	27	187	402	327	143	
228	45x	28	28	28	355	368	447	171	
229	45x	87	77	29	194	404	358	494	
230	66x	00	00	30	004	006	006	006	COLUMN SUM CARDS
231	799	83	73	31	812	666	047	218	
232	888	00	00	00					SPACER CARD

Fig. 6

```
        CAUTION:  CLEAR COUNTER GROUPS AND AUX. STORAGE ON FINAL TOTAL BEFORE RUNNING

              ROW 11 TO START
 181
 476                       181
 394                       476
 292                       394
 232                       292
 142                       232
 281                       142
 159                       281
 437                       159
 225                       437
 495                       225
 484                       495
 163                       484
 365                       163
 122                       365
 114                       122       MINIMUM ROW ELEMENT
 248                       114  <--- INDICATES COL 25 TO BE PLAYED NEXT
 458                       ---
 340                       248
 153                       458
   5                       340                                    LINE
 761                       153
       SUM OF ELEMENTS OF ROW 11 ---------> 5761  114.00        1
                                                  V
                                                   n
COLUMN 25        CHANNEL B
                                    CHANNEL C
 240     181                                             25
 114     476                 181      240
 426     394                 476      114            COLUMN BEING PLAYED
 118     292                 394      426
 453     232                 292      118
 103     142                 232      453
 498     281                 142      103
 333     159                 281      498  <--- MAXIMUM COLUMN ELEMENT
 408     437                 159      ---       INDICATES ROW 16 TO
 195     225                 437      333       BE PLAYED NEXT
 475     495                 225      408
 196     484                 495      195
 439     163                 484      475
 303     365                 163      196
 339     122                 365      439
 167     114                 122      303
 209     248                 114      339
 413     458                 248      167
 150     340                 458      209
 205     153                 340      413
   5                         153      150
 784                                  205
              5761
              SUM OF ROW SUMS SO FAR          5784  498.00       1
                                                    _
                                                    V
                                                     n
      SUM OF ELEMENTS OF COLUMN 25

                              Fig. 7
```

ROW 16	CHANNEL B		CHANNEL C	
486	181	240		
306	476	114	667	240
375	394	426	782	114
366	292	118	769	426
257	232	453	658	118
145	142	103	489	453
430	281	498	287	103
353	159	333	711	498
379	437	408	512	333
239	225	195	816	408
211	495	475	464	195
428	484	196	706	475
444	163	439	912	196
132	365	303	607	439
258	122	339	497	303
498	114	167	380	339
301	248	209	612	167
324	458	413	549	209
371	340	150	782	413
274	153	205	711	150
			427	205

16 ROW BEING PLAYED

MINIMUM ROW SUM INDICATES COLUMN 15 TO BE PLAYED NEXT

6
577
SUM OF ELEMENTS OF ROW 16

5784
SUM OF COLUMN SUMS SO FAR

12338
SUM OF ROW SUMS SO FAR (= 5761 + 6577)

143.50
$\underline{V}_n$

LINE
2

COLUMN 15				
369	667	240		
142	782	114	667	609
389	769	426	782	256
200	658	118	769	815
175	489	453	658	318
414	287	103	489	628
145	711	498	287	517
201	512	333	711	643
403	816	408	512	534
391	464	195	816	811
137	706	475	464	586
474	912	196	706	612
110	607	439	912	670
168	497	303	607	549
335	380	339	497	471
284	612	167	380	674
491	549	209	612	451
253	782	413	549	700
411	711	150	782	666
381	427	205	711	561
			427	586

15 COLUMN BEING PLAYED

INDICATES ROW 12 TO BE PLAYED NEXT (MAXIMUM COLUMN SUM)

5
873
SUM OF ELEMENTS OF COLUMN 25

12338
SUM OF ROW SUMS SO FAR

11657
SUM OF COLUMN SUMS SO FAR (= 5784 + 5873)

407.50
$\overline{V}_n$

2

R - ______ Section ____

STEP # ______ . Keypunch ____ cards from the attached manuscript.

STEP # ______ . Second operator KP cards from attached manuscript.

STEP # ______ . Compare and correct, saving both corrected decks.

STEP # ______ . Adjoin both decks and duplicate to handful size. Place in front of this handful a starter card, with a 1 in col 6 and XY in col 74, and a blank. Wire a 521 panel to summary punch (see page 4 of CSM 315). Run the deck on the games board with Setup Change Switch #1 OFF. Continue runs until either the left hand or right hand five digits of Channel B overflow to 100,000. Start with a row choice of ____.

CARD FORM:

Choice	Line	$\underline{V}_n$ or $\overline{V}_n$
1 2	3 4 5	6 7 8 9 10

20 x 20 GAME MATRIX — KEYPUNCH MANUSCRIPT

SER	OPER	B	C	N	10	20	11	21	12	22	13	23	14	24	15	25	16	26	17	27	18	28	19	29	X_Y in col
110	235	84	740	10																					
111	35X	11	110	11																					
112	35X	12	120	12																					
113	35X	13	130	13																					
114	35X	14	140	14																					
115	35X	15	150	15																					
116	35X	16	160	16																					
117	35X	17	170	17																					
118	35X	18	180	18																					
119	35X	85	750	19																					
120	35X	86	760	20																					
121	35X	21	210	21																					
122	35X	22	220	22																					
123	35X	23	230	23																					
124	35X	24	240	24																					
125	35X	25	250	25																					
126	35X	26	260	26																					
127	35X	27	270	27																					
128	35X	28	280	28																					
129	35X	87	770	29																					
130	66X	00	000	30																					80
131	799	82	720	31																					74
132	888	00	000	00																					
210	245	84	740	10																					
211	45X	11	110	11																					
212	45X	12	120	12																					
213	45X	13	130	13																					
214	45X	14	140	14																					
215	45X	15	150	15																					
216	45X	16	160	16																					
217	45X	17	170	17																					
218	45X	18	180	18																					
219	45X	85	750	19																					
220	45X	86	760	20																					
221	45X	21	210	21																					
222	45X	22	220	22																					
223	45X	23	230	23																					
224	45X	24	240	24																					
225	45X	25	250	25																					
226	45X	26	260	26																					
227	45X	27	270	27																					
228	45X	28	280	28																					
229	45X	87	770	29																					
230	66X	00	000	30																					80
231	799	83	730	31																					74
232	888	00	000	00																					

R - ________ Section ____

Number of lines played =

STEP #_____ . Having completed the required number of lines on step #___, take an even number of summary cards (row and column choice for each line) and order the deck on $\underline{V}_n$; split the deck in half by card count. The lower half deck contains $\underline{V}_n$ and column choice, the upper half $\overline{V}_n$ and row choice. Fill in:

$\underline{V}_n$ max = ________ at line # ____; $\underline{V}_n$ 2nd max = ________ at line # ______

$\overline{V}_n$ min = ________ at line # ____; $\overline{V}_n$ 2nd min = ________ at line # ______

These four values will be found in the two cards on either side of the break in the deck. Sort each half deck on line number. Using the sorter, sum the row choice frequencies thru this line, then thru this line , then thru all the lines. Now sum the column choice frequencies thru this line , thru this line , and then thru all the lines. Tabulate in chart below.

CHOICE	ROW CHOICE FREQUENCY						COLUMN CHOICE FREQUENCY					
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												
26												
27												
28												
29												

CARD FORM: Row or column choice in columns 1 and 2
Line number in columns 3, 4, 5
$\underline{V}_n$ or $\overline{V}_n$ in columns 6 to 10

5

PROCEEDINGS

OF THE SECOND SESSION

- GUIDE -

February 13-15, 1957
Conrad Hilton Hotel
Chicago, Illinois

391

TABLE OF CONTENTS

Appendix Z

REPORT ON HISTORY AND USE OF AUTOMATIC CODING

MR. R. W. BEMER of IBM: This business of the coding systems cannot be covered in small talk such as this. It is an unholy monster and has been going on for a number of years. Like many field of knowledge, it has to divert itself greatly before we can cut it down to size again.

Some of you here have been in this business for quite a bit of time and are familiar with many of the things I would like to say here. Others of you, I am sure, are not too familiar with computing systems outside of the 702 and 705 and perhaps may not be aware of some of the history and some of the reasons these things came into being.

Of course, it has not always been as good as it is today, and, furthermore, what we now think is a good thing is terrible, it is going to be improved in the future.

Inasmuch as I can cut this thing down, I would like to say that many of these things I would have otherwise talked about here can be found in some articles in Automatic Control magazine for March and April and I think will provide some means of keeping this information on tap. More or less the history of this business and all these articles are directed towards the engineers but then I believe it applies equally to people in the commercial field.

I have a rather large list here which you certainly are welcome to look at later on. I have some 83 automatic coding systems which are completed. The first of this was established in February of 1951 and, as I recall that date, things at that time were in pretty sad machine language at best.

The machine language at best is a poor language to work with for it does not have any relationship, that I can find, to the program of solving language and, if we never see it again, it will be too soon. I hope that you feel the same way that I do because most of you are in business to solve problems of one description or other.

I don't think that you should have any concern with the computer itself. I think that you should be able to state your problem in connection with any particular model and in a language that the computer will work for you.

When you speak of the machine language, this means our lowest level, the only language which the computer will understand. We realize that we code problems now to do work, that the automatic coding systems are further extensions of machine usage and so they will take over still another facet of our work. In other words instead of doing quickly the problems that used to take a lot of handwork and comptometer work, the machine takes over in the same way and does a good deal of the repetition in programming which we formerly had to do. This is undoubtedly very fine but then it is not the end by any means.

The next step up beyond that will be to provide learning and intelligence in these programs and also let us guys that have to worry about making these automatic coding systems take advantage of the machine also, and I will describe how this might be done.

I think those of you who have worked with machine language, with the 705, symbolic coding and so on, have enough experience in this so that you are aware of the great savings you can make in manpower. This computing business has been growing, at least ever since I have been in it and if we keep on doubling people or programming every year, pretty soon the entire population of the United States is going to be doing programming. Therefore, to eliminate the bottleneck, we have to get good automatic coding systems going, far superior to the ones that we have now.

I think that this will probably effect, within the next two years, savings on the order of ten to one, wherein one programmer will be able to do the work of ten and, if we can also do some other things we have in mind, we will be able to make savings on the order of one hundred to one. This will bring us down more to the realm of possibility.

In order to bring this thing down, I think that we have got to be within a ratio of about fifty per cent of what we can do with automatic coding systems as they exist now. We have got to write equations for operations, the various statutes that are applicable and so on. There, even if we get a very fine language with which to describe our problem, it is going to be very difficult to cut it down too much more than we have now. It is true that we will put it into English and that will make savings, but the processing is still quite large.

What I would like to quickly tell you about is some of the plans we have over at IBM to make this large savings for the future. The reason that I say this is because the very name of your organization, "Guide" should enable you to be guided by some of the things that I would like to tell you. I do not want to worry you about them but I would like to have you know as to what may be coming up in the future and slant your thinking in that direction.

The program, after it is written, is essentially a depository of the intelligence of the man that writes it. If you are doing a hand calculation program, you can go through the thing because you have made the calculations and you know how to do it. However, suppose that someone else wants to do it and you have to teach them how to do it. If 500 want to do it then you have to essentially teach them in a class or teach ten, who, in turn, go out and teach the rest.

With programming on the computer this is not necessary any more. The program, as you write it, has input and output from that point on and, after that, you do not have to know what goes on inside. All that you have to know is what you put in and what you want to get out on the basis of such and such a process.

Now then, if we can do this sort of thing, we have reduced a good deal of our labor. Let me tell you how this will help us in the future.

Two hundred years ago mankind was saddled with roughly a fifth of a horsepower and books were written at that time to prove that it was impossible to have more than that. However, as we know, we have increased that so that there is available, to every person in the United States, roughly 200 horsepower at his own personal command. This is in line with the cars and other mechanical devices we have in this country. We have extended our power here and it hasn't seemed to do us any harm.

It is now to the point where things have progressed to the point of where they are pretty much beyond us. No one man is smart enough to take care of our social and economic problems any more and the only solution is to expand our brain power and we do this by linking together various things. I believe that we can do this by linking together intelligence and knowledge of the people themselves in the problems.

Obviously there must be some correlation between these things and so this is what we have to do. We must devise learning programs for the computers. I realize that there are many learning programs of a simple nature. I now have reference to the work of Simon and Newell. They have developed a logic machine that I think is going to have a lot of implications. It is a computer but, instead of computing what you might consider payroll, they program a logic.

The first problem they set the program to doing was proving the mathematical theorems. I would like to illustrate that on the board by the use of the normal triangle. This is a triangle and we have given here equal angles and the problem is to prove that the opposite sides are equal. Of course, we all know the technique used to do this. However, the machine would do it differently. The machine has certain learning mechanisms in which it alters its own program so that a good programmer cannot predict what will finally be in the machine. Here is what the machine does. It says "I have a side angle and it may intersect. I know this by an angle side angle combination and, therefore, I flip it over on itself. I will take a mirror image and drop it. Therefore, it is on its side, superimposed on the same place. The angle on the right is now the same angle on the left and so it must be the same triangle. Now the side that used to be on the right hand side is occupying the left hand position and the left hand side is now on the right and so they must be equal". This is a much more intelligent proof than the mathematics involved. Of course, this is only the start of these things. We can reconstruct exactly how the machine thought as it went around doing these things.

Therefore, what we are doing now is trying to find learning procedures such as this, whereby the machine may alter itself on program. It not only gives you a better and more efficient program but it will do things that we as a single person could never have thought of before.

In the programming research of IBM, we have approximately 30 people right now and we expect to have up to 75 by the end of the year. This recognizes just one thing, that a system for using a computer is as important as the computer and, furthermore, they must both roll off the line together in order to have a good system.

Therefore, IBM is committed to producing these systems and we are presently working on some better automatic coding languages which we think will be far superior to anything we have right now. Of course, it is going to take a few years before we can get them to you.

I think that the interesting thing will be this, that you will not have to worry about switching. At this time we think that we are grown up enough in this business to look ahead and we are designing this language so that it will work on the most advanced computers.

We want to have a supervisory operation so that the machine schedules itself, figures out what problems should be done next. If this is done then we feel that we will then have a good means of making various programs together.

Furthermore, if I have sequence to do, I like to make my accounts payable and payrolls. If I have done payroll 34 and accounts payable 28 and various things, I would like to be able to say, "Do these in this order. Accounts payable; payroll; Jones, Smith, Pete Brown". You have all these names in very, very large operations and you will find, I am sure, that your thinking with respect to both commercial sides of the prolems will be greatly facilitated if you keep this method in your head, and I expect many of you do that with profit there.

IBM has expended a great deal of effort on making this language. I think so much that probably any one installation or even group installation will not be able to afford it or profit by doing this particular thing any more. Before the transition of 705, I know of exactly one person who has a fair idea of what goes on all through the system. The people that did the components did not know anyone else's part very well, and this is only our present system. The system of three of four years in the future will be such that it will be probably worked on by forty people and not one of them will realize what goes on in anybody else's section, and then when you find these learning techniques and the machine statistically improves its own program, nobody, even the man who originally wrote it, will be able to recognize it.

So, it is obvious that the manufacturer has the responsibility to produce and use these systems and he will have to do it and it will take plenty of manpower.

Now, I don't like to hire any more than the next person, so I would like to leave you with something in mind.

We profited by having a man from United Aircraft at Hartford and also a man from the University of California Radiation Lab work on this' while it was developed. They did three things by this. They helped the formulation and completion of the system. They learned the inside of the system to a much greater extent than the normal user, so if they wanted to make special variations for their own usage, they were in much better shape. The third thing was, they got to see some of the other people in the business and shared and cross-pollinate ideas and information.

Now, I would like to propose that something like this might be possible for the languages that IBM is going to develop now. We have a woman from Bell Telephone Laboratories now working on the Fortran systems with the 32,000 word 704, and this apparently, so far, has worked out very well for both the people concerned and the general industry, because of the way we submit our system.

It will be to your advantage to contribute the man, if you have him to spare, or have their services available, to use on a consulting basis, so we can try to work the language out. Be our guinea pig. Would you mind learning the language and trying it on a theoretical problem or one you have in your shop? Only in this way will we know how to make the common, pure language that is applicable to everyone's purpose and usages. Thank you.

Why the Engineer Should Know Computer Programming

One of the major expenses of setting up a modern high-speed computer to do useful work is incurred by programming—the science of translating a problem to terms and instructions that the computer can understand and obey.

The type of problem affects programming costs ranging from 60 to 100% of the cost of the computer itself, which varies from approximately $30,000 to more than $2,000,000 in production models. If a program is poorly constructed, the running program can take many times as long as it normally should to produce the desired answers.

With a trend now developing toward giving engineers more direct access to computers, efficient use of these modern engineering tools charges them with new responsibilities. The concluding article next month will detail *what* the engineer should know about programming.

BY ROBERT W. BEMER
Programming Research Dept.
International Business Machines

■ The engineer knows his own problem best. Simple economy dictates that it is better for the engineer to learn ONCE how to program any problem for the computer, rather than his explaining each new problem to the professional programmer with resultant loss of time and effort.

The man with the problem may also make decisions "on-line," during the course of the computer run, from the physical implications of the answers as they are produced. These fine gradations of magnitude and interrelation are immensely more significant to the engineer, whereas the operator can seldom be instructed to properly make these often delicate decisions. The direct-user technique can greatly reduce the computational costs of certain types of problems by eliminating calculations along visibly fruitless lines of investigation.

Actually there are two ways to reduce costs of computer operations. One is to reduce the actual labor spent in programming, affecting salaries and overhead. The other is to make the program more efficient from the machine standpoint, thus minimizing operating costs in dollars per problem solved. Both of these are of vital interest to the engineer because of the worth of his time. Furthermore, management must be kept convinced that this profitable tool actually makes money for the company as well as reducing the complexity of the engineer's tasks. These factors contribute to the many sound reasons why the engineer should know and understand computer programming.

Computers permit cooperative efforts between engineers which minimize the work each must do. When properly constructed, computer programs are "open-ended" and allow refinements and additions to be appended at any time. After seeking competent advice, the engineer inexperienced in programming should make a modest start on a single portion of his problem, which may then be augmented as planned or as initial results demonstrate to be desirable. For library purposes, the program may then be considered to be the repository of the intelligence of the engineer concerning that particular problem.

Furthermore, a computer program tends to clarify and organize a problcm much as explaining it to another person does, except that it is less gullible. When properly named, this program is now available as a component in a larger problem. If a hand-calculation is performed, only the results of that specific case remain; the method itself may not be distributed to others except by a teaching process. When coded for a computer, however, it is available to everyone without regard to the internal process. It essentially becomes a "black box" and all the user must know are the specifications for the input and output. By extension, it is possible for a group of engineers to unify the whole spectrum of their work. This complex of programs now represents a unified system, although programmed switches are usually inserted so that certain portions of the calculation may be bypassed when not required for a specific application.

Thus, the computer affords the engineer the long-coveted opportunity of shedding the drudgery of numerical calculation in its most repetitious forms. If the computation essential to a certain class of problems is reduced to a generalized form which automatically produces correct answers merely upon specification of controlled input, then the arithmetic-bogged engineer is free to do engineering in the true and creative sense.

Vast engineering experience can be gained in a minimum time with a computer. Many of us know the old hand who can predict the exact performance of a new airplane, the precise way to design a boiler, or the exact proportions for the most efficient bridge design. Few of these men are born; most of them achieved such abilities by intelligent correlation of the cross-effects of many thousands of variations in design, observed through the many years of their experience.

A computer can condense this experience in time scale, processing many thousands of variations in a short time once the controlling conditions and formulae have been specified. For example, instead of designing a single airplane and completing the analysis slightly before the prototype is built, aircraft engineers now use computers to try hundreds of designs. They may make the final decision and design selection on comparative costs as well as performance. Then too, many problems are now solvable for which there does not exist a classical method and so were only roughly approximated heretofore. One would hardly use a rigorous method to solve a cubic equation on a computer; it is possible to solve a 50th

Reprinted through the courtesy of AUTOMATIC CONTROL, February 1957

order equation by iteration in not much more time than it takes to push the computer START button.

The engineer may find unexpected sources of computing power in his company. It is quite common for computers to make their advent at a company through the accounting or production control departments. However, the engineer who is cognizant of the characteristics of computers and programming is also aware that computers originally designed for doing commercial work are capable also of doing engineering work, and vice versa. Most computer manufacturers provide relatively easy programming systems for performing these dual roles.

It is important for the engineer to know how to justify computers for his needs, and in what pattern the work load should expand. Certainly any computer should go into reasonable production to earn its keep from the moment it is installed. Even with the most enlightened management it is difficult to properly explain the amount of preparation and programming which must be done in advance of delivery. This is additionally complicated by the axiomatic condition that while the most efficient machine for the engineer is the largest and most expensive, it is the most difficult to initially load and justify.

TYPICAL ENGINEERING APPLICATIONS FOR COMPUTERS

Until the engineer actually starts to investigate the programming process he is not likely to be aware of all the opportunities for a computer to serve him in his work. This list of typical existing engineering applications should prove to be a useful guide.

AERONAUTICAL ENGINEERING

Aeroelastic, utter and vibration analysis
Armament systems evaluation
Bombing systems evaluation
Body and duct design, lofting
Compressible flow studies
Data reduction-telemetered, theodolite, wind tunnel
Engine cooling
Fire control pursuit course calculations
Flight trajectory calculations
Fuel cell pressure analysis
Guidance problems
Guided missile optimization studies
Heating studies
High-speed instrumentation
Landing gear design
Load, shear and moment calculations
Nozzle design
Optical system design
Power plant performance calculations
Radar equipment design
Radar detection probabilities
Radar echo studies
Radio interference
Radome studies
Servomechanism calculations
Sound pressure analysis
Standard performance calculations
Wind tunnel balance computing

CHEMICAL ENGINEERING

Absorption analysis
Crude oil evaluation
Flash vaporization
Gas vapor cycle performance coefficient
Liquid-vapor equilibrium calculations
Mass spectrometer analysis
Multi-source planar diffusion
Pilot diffusion cascade data analysis
Pipeline design, stress analysis
Refinery simulation, production analysis
Tankage studies

MATHEMATICS

Algebraic equations—real and complex
Applied probability functions
Complex polynomials
Eigenvalues
Fourier analyses
Generation, tables of special functions
Linear programming
Matrix calculations
Minimize functions of two variables
Ordinary differential equations
Random-number generation
Random walks
Simultaneous linear and non-linear equations
Simultaneous linear and non-linear differential equations
Transportation problems

ELECTRICAL ENGINEERING

Circuit design and minimization
Circuit breaker design
Motor and generator core losses
Motor and generator—critical shaft speeds
Power system—economic operation
Power system—loading and losses
Power sub-station studies
Stability and transient studies
Transformer design

PHYSICS

Atomic power studies
Gamma ray attenuation
Neutron absorption breakdown
Nuclear calculations
Upper atmosphere research
X-ray crystal structure analysis

STATISTICS

Analysis of variance
Auto-correlation and power spectra
Climatological statistical analysis
Least squares curve fitting
Multiple correlation and regression
Multiple bivariate frequency distribution tables of weather elements
Quality control
Standard deviations and means

MISCELLANEOUS

Bridge and truss design
Traffic control
Cut and fill—road-building

Before the advent of automatic coding systems, which relegate to the computer itself most of the work caused by the nature of the machine language, there was an "open-shop" versus "closed-shop" controversy in the computing field. Programming for a computer was a difficult and tedious art to learn, with many "housekeeping" functions to be performed again and again. Unfortunately, these functions were caused by the limitations of computers; they contributed nothing to the solution of the problem. Most computer-equipped companies leaned to the closed-shop, teaming a programming specialist with the engineer because they felt it was too difficult and expensive to teach programming to all of their engineers.

Although many inefficiencies were thus created, a few companies pioneered the open-shop and we are in their debt for the methods that they developed and for forcing the automation of coding. Today the controversy is simply settled. Available automatic coding systems (to be completely listed in the first published directory of them next month—Ed.) now make it easy and worthwhile for the engineer to do his own programming in a "problem-solving" language rather than a "machine" language, thus fully realizing the benefits of the open-shop. All closed-shop people now concentrate on fabricating the much more intricate and intelligent automatic programming systems of the future. ■ ■

What the Engineer Should Know About Programming

How to Consider A Computer

Engineering is taking on a "new look." Computers are the logical and more powerful successors to the desk calculator and the slide rule, the previous working tools of the engineer. There is really only one major difference: because of their necessary size and cost to be so powerful, computers must be shared by a great many users. This means a new concept of shared system operation must be accepted by the engineer.

To help you get oriented, here are some vital considerations affecting present and future computer use in your work and some helpful sources of further highly specialized information.

BY ROBERT W. BEMER
Programming Research Department
International Business Machines

■ A computer should not be rented or purchased unless an automatic programming or coding system is furnished for its operation. The computer and the operational system constitute a matched pair, and one without the other is highly unsatisfactory from the point of view of getting work done at minimum cost.

For engineering work, any automatic system should contain provision for indexing and floating point operation, if these are not built in as hardware, for they are the two most vital features for easy usage. Indexing allows for algebraic array notation, which in turn makes for easy understanding of how a problem should be programmed. Floating point, although it may sometimes introduce either spurious accuracy or loss of it to the uninitiated, prevents a Gordian tangle of scaling difficulties from cluttering up the problem.

HOW CODING SYSTEMS HELP

Automatic coding systems have by no means reached their ultimate efficiency or sophistication, yet remarkable savings in programming costs have already been achieved, sometimes by an order of 50! For the best of the present systems it is a reasonable estimate to say that they can, in general, reduce the programming costs and time to a tenth of that required to code in stubborn machine language.

There have been many attempts to relieve the burden of programming through special coding systems of all types. The data sheet on computer coding systems is not only an interesting history of growth, but is also presented for the edification of those now entering the field with incomplete knowledge of what code to use for their machine. The time may come soon when you will be using a common language exclusive of the characteristics of any particular computer. Thus, with an automatic translator for each different computer, a running program may be produced for any desired machine from the single original problem and procedure statement in the common language. Credit is due to Dr. Saul Gom of the Moore School of Electric Engineering for first championing these principles.

GOOD COMPUTER OPERATION IS STATE OF USER'S MIND

It is axiomatic that a computer should never stop, run useless problems or be subjected to manual oper-

ation and dial-twiddling. To do so deprives your fellow engineers of its benefit. Here are some detailed considerations pertinent to good computer operation:

- It can do only what it is explicitly ordered to do, and this ordering must be done eventually in its own machine language, which is all it can understand.

- The reliability of most present-day computers is so high that answers are not right or reasonable, the chances are at least 99 to 1 that it is somehow the user's fault. Wrong answers usually stem from wrong equations or misuse.

- Allow for growth when doing the original planning. Build in flexibility for changes, or else costs will soar if the entire problem must be re-programmed. A stored-program may always be corrected or augmented to give exactly what the engineer desires, including special report format for jobs where repetition justifies the effort.

- For design studies, plan parameter variation carefully and allow flexibility in changing individual parameters. The computer may surprise you by showing that some parameter values for optimum conditions may be outside of the range expected or allowed for. To make certain that the program gives answers consistent with hand-computations, first test the program on the machine for a specific combination. Time this run. Then multiply this time by the total number of parameter combinations to see if the study is feasible in time and cost. If M parameters are combined for N values each, the total number of combinations is N^M.

For example, with 6 Parameters: 6 values for each will produce 46,656 combinations; 5 will produce 15,625; 4 will produce 4,096.

The moral: *Don't triple the cost* of your problem if you are engineer to draw a curve through one less point.

FUTURE COMPUTER LANGUAGES

New synthetic languages are in the process which will affect your use of computers. As problem-solving languages they will be much superior to present systems in these ways:

1. Even though the binary type of computer will probably be universal for both engineering and commercial work, the need for the user to know binary representation will effectively disappear. Logical decision will be the only remaining function which will not appear in decimal form, and even here facilities will be provided so that the programmer need not concern himself with the precise method of operation within the machine. Conversion from fixed point decimal to floating point binary for operation, and the converse for output, will all be done automatically by the synthetic language translator.

2. The elements of the synthetic language will be essentially algebraic, both arithmetic and logical, and linguistic so that procedures may consist of real sentences in a living language. Idiom will be such that the program will be operative in any spoken language, with minor changes

COMPUTER	SYSTEM NAME OR ACRONYM	DEVELOPED BY	CODE	SYSTEM TYPE				OPER. DATE	INDEXING	FL PT	SYMB.	ALGEB.	COMMENTS
				M.L.	ASSEM.	INTER.	COMP.						
I.B.M. 704	R-S	Los Alamos			X			Nov. 55	M2	M	1		
	Cage	General Electric			X		X	Nov. 55	M2	M	2		
	Fortran	I.B.M.	1				X	Jan. 57	M2	M	2	X	
	NYAP	I.B.M.			X			Jan. 56	M2	M	2		
	Pact 1A	(See Pact Group)					X	Jan. 57	M2	M	1		Modified Pact I for 704
	SAP	United Aircraft	1		X			Apr. 56	M2	M	2		Official Share Assembly
Sperry-Rand 1103A	Compiler I	Boeing, Seattle			X		X	Mar. 57		S	1	X	
	FAP	Lockheed M.S.D.	1	X		X		Oct. 56	S1	S	0		
	Mishap	Lockheed M.S.D.			X			Oct. 56		S	1		Magn. Tape Assembly + Correction
	Trans-Use	Holloman A.F.B.			X			Nov. 56		S	2		
	Use	Ramo-Wooldridge	1		X		X	Feb. 57	M/S1	M/S	2		Official for Use Organization
1103	Chip	Wright Field		X		X			S1	S	0		Similar to Flip
	Flip/Spur	Convair San Diego		X		X		Jun. 55	S1	S	0		Spur Unpacked, Twice as Fast
	Rawoop	Ramo-Wooldridge	1		X			Mar. 55	S1	—	1		One-Pass Assembly
	Snap	Ramo-Wooldridge	1		X	X		Aug. 55	S1	S	1		Used with Rawoop
I.B.M. 705	Autocoder	I.B.M.	1	X	X		X	Dec. 56	—	S	2		
	Fair	Eastman Kodak				X		Jun. 56	—	S	0		
	Print I	I.B.M.	1	X	X	X		Oct. 56	S2	S	2		
	Symb. Assem.	I.B.M.			X			Jan. 56	—		1		May be Assembled on Accng. Equip.
	SOHIO	Std. Oil of Ohio		X	X	X		May 56	S1	S	1		
Sperry-Rand Univac I, II	A0	Remington Rand		X	X		X	May 52	S1	S	1		
	A1	Remington Rand		X	X		X	Jan. 53	S1	S	1		
	A2	Remington Rand		X	X		X	Aug. 53	S1	S	1		
	A3	Remington Rand	2	X	X		X	Apr. 56	S1	S	1		
	AT3	Remington Rand	2		X		X	Jun. 56	S1		2	X	Fortran-Like, Output To A3, I + II
	BO	Remington Rand	1,2	X	X		X	Dec. 56	S2	S	2		Runs on Univac I + II
	BIOR	Remington Rand		X	X		X	Apr. 55	—	—	1		Primarily Business Data-Processing
	GP	Remington Rand	1	X	X		X	Jan. 55	S2	S	1		For Expert Programmers
	NYU	New York University					X	Feb. 54	—	S	1		
	Relcode	Remington Rand		X	X			Apr. 56	—	—	1		
	Short Code	Remington Rand		X		X		Feb. 51	—	S	1		
	X–1	Remington Rand	2	X	X			Jan. 56	—	—	1		Runs on Univac I + II
I.B.M. 702	Autocoder	I.B.M.		X	X		X	Apr. 55	—	S	1		
	Assembly	I.B.M.			X			Jun. 54	—	—	1		May be Assembled on Acctg. Equip.
	Omnicode	G.E. Hanford	2		X	X	X	Propos	S2	S	2		Super-Script
	Script	G.E. Hanford	1	X	X	X	X	Jul. 55	S1	S	1		
I.B.M. 701	Acom	Allison G.M.				X							
	Bacaic	Boeing, Seattle	1,2		X		X	Jul. 55	—	S	1	X	
	Douglas	Douglas Sm			X			May 53	—	S	1		
	Dual	Los Alamos		X		X		Mar. 53	—	S	1		
	607	Los Alamos			X			Sep. 53	—	—	1		
	Flop	Lockheed Calif.		X	X	X		Mar. 53	—	S	1		
	Jcs 13	Rand Corp.			X			Dec. 53	—	S	1		Modification of 607
	Kompiler 2	Ucrl Livermore					X						
	Naa Assembly	N. Amer. Aviation			X								Also Assembles 704 Programs
	Pact I	(See Pact Group)	1				X	Jun. 55	S2	—	1		Most Programs run on Pact IA
	Queasy	Nots Inyokern				X		Jan. 55	—	S	—		
	Quick	Douglas Es				X		Jun. 53		S	0		Double Quick for Dbl Prec
	Seesaw												
	Shaco	Los Alamos			X	X		Apr. 53		S	1		
	So 2	I.B.M.			X			54					
	Speedcoding	I.B.M.	1		X	X		Apr. 53	S1	S	1		
Datatron	Ctc	Purdue Univ.	2				X	Incomp	S2	S	1	X	
	Dot I	Electro Data				X							
	Ugliac	United Gas Corp.				X							
I.B.M. 650	Ades II	Naval Ordnance Lab					X	Feb. 56	S2	S	1	X	
	Bacaic	Boeing, Seattle	2		X	X	X	Aug. 56	—	S	1	X	Must Process on 701
	Balitac	M.I.T.		X	X		X	Jan. 56	S1		2		For all 650 models
	Bell	Bell Tel. Labs		X		X		Aug. 55	S1	S	0		
	Ctc	Carnegie Tech	1,2				X	Dec. 56	S2	S	1	X	Output Processed by soap
	Druco I	I.B.M.				X		Sep. 54		S	0		
	Flair	Lockheed Msd, Ga.		X		X		Feb. 55	S1	S	0		
	Mitilac	M.I.T.		X		X		Jul. 55	S1	S	2		For all 650 models
	Omnicode	G.E. Hanford	2			X	X	Dec. 56	S1	S	2		Must Process on drum 702
	Sir	I.B.M.				X		May 56	—	S	2		Operates with soap I, II
	Soap I	I.B.M.			X			Nov. 55	—		2		
	Soap II	I.B.M.	1		X			Nov. 56	(M)	(M)	2		For all 650 models + variations
	Speed coding	Redstone Arsenal		X		X		Sep. 55	S1	S	0		Resembles 701 speedcoding
	Spur	Boeing, Wichita		X	X	X		Aug. 56	(M)	S	1		For all 650 models
Whirlwind	Algebraic	M.I.T.					X		S2	S	1	X	
	Comprehensive	M.I.T.		X	X	X		Nov. 52	S1	S	1		
	Summer ses.	M.I.T.				X		Jun. 53	S1	S	1		
Midac	Easiac	Univ. of Michigan			X	X		Aug. 54	S1	S	1		
	Magic	Univ. of Michigan			X	X			S1	S	1		
Burroughs		Burroughs Lab					X				1		
Ferranti	Transcode	Univ. of Toronto	1		X	X	X	Aug. 54	M1	S	1		
Illiac	Dec order input	Univ. of Illinois	1		X			Sep. 52	S1	S	1		
Johnniac	Easy fox	Rand Corp.			X	X		Oct. 55		S	1		
Norc		Naval Ordnance Lab			X		X	Feb. 56	M2	M	1		
Seac	Base 00	Natl. Bur. Stds.		X		X							

Pact Group Contains Douglas Sm, Es, Lb, Lockheed Cal, Nots, N. Amer., Rand in the dictionary, much as the FORTRAN coding system has already been translated for use by the French.

Much more intelligence will be built into the translators. The program may make a reasonable guess about the intent of the programmer when some omission or violation of language rules occurs. Learning procedures will be incorporated so that the translator may take advantage of statistics of previous operation to improve the program which it creates. This may extend as far as a form of creativity where the date processor may originate programs merely from the statement of the problem to be solved, rather than from a given procedure for this solution.

4. Flow-chart construction for procedures will be automatic, having been determined in their linkages by the before-and-after conditions for the components of procedure. Today, Monte Carlo techniques are already being used to determine frequencies of occurrence for various logical branchings in the procedure, with resultant optimization of the program by taking these factors into consideration. The efficient usage of the various and graded components of the computer system will be automatic; the programmer considers an infinite machine, which the processor arranges as it knows its own limitations and capabilities.

5. Not only will programs be created which write programs to do actual computation, but other levels will be superimposed on these. Bootstrap methods are being considered which will allow even the person who develops the processor to have a good portion of his work done automatically by reference to and through previous work.

EXPLANATION OF SYMBOLS

CODE 1) Recommended for this particular computer, sometimes for volume of usage or interchange, or for lack of better.
2) Common language to more than one computer.

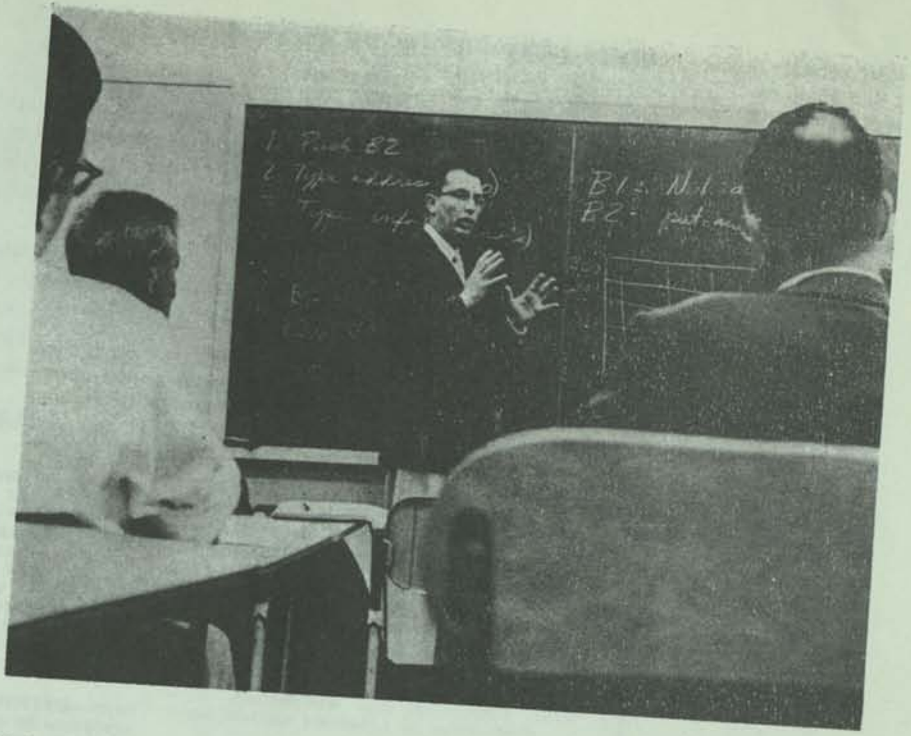
SYSTEM TYPE (See "Terms Used In Automatic Coding")

INDEXING M) Actual index registers or B-boxes exist in machine hardware.
S) Simulated in the synthetic language.
1) Limited form, either stepped unidirectionally or by 1 word only, having certain registers applicable to only certain address positions, or not compound (by combination).
2) General form, where any address may be indexed by any 1 or a combination of registers, which may be freely incremented or decremented by any amount. Have associated loop termination instructions.

FLOATING PT. M) Inherent in machine hardware.
S) Simulated in the synthetic language.

SYMBOLISM 0) None.
1) Limited—either regional, relative or exactly computable.
2) Fully descriptive, where a word or symbol combination may be descriptive of the contents of the assigned storage.

ALGEBRAIC A single continuous algebraic formula statement may be made. The processor contains mechanisms for applying the associative and commutative laws of algebra to form the operative program.



6. The actual operation of the computer will be under control of an integrated portion of the processor known as a supervisory routine. In some instances the program will not have been created by the processor prior to execution time, but will be created during a break in execution time under orders of this supervisory routine, which detects that no method is in existence in the program for a particular contingency. Although these supervisors will be on magnetic tape for a while, it is envisioned that they will be buried in the machine hardware eventually, to be improved by replacement like any other component.

FUTURE COMPUTER SYSTEMS

Future computer operation, which strongly influences the design of the programming languages, has some vitally interesting possibilities. In this glimpse, the picture presented here is dependent upon three axioms:

▶ Faster computers always lower the dollar cost per problem solved, but not all companies will be able to afford the high prices of the next generation of super-computers. They simply may not have enough problems to load one.

▶ Producing a spectrum of machines is a tremendous waste of effort and money on the part of both the manufacturers and the users.

▶ Availability of a huge central computer can eliminate the discrete acquisition of multiple smaller computers, homogenize the entire structure of usage, and allow a smaller and more numerous class of user into the act, thus tapping a market many times the size of presently projected with current practice in computer access.

Assuming the availability of practical micro-wave communication systems, it is conceivable that one or several computers, much larger than anything presently contemplated, could service a multitude of users. They would no longer rent a computer as such; instead they would rent input-output equipment, although as far as the operation will be concerned they would not be able to tell the difference. This peripheral equipment would perhaps be rented at a base price plus a variable usage charge on a non-linear basis. The topmost level of supervisory routine would compute these charges on an actual usage basis and bill the customer in an integrated operation. These program features are, of course, recognizable to operations research people as the Scheduling and Queuing Problems.

Using commutative methods, just as motion pictures produce an image every so often for apparent continuity, entire plant operations might be controlled by such super-speed computers.

These future hardware capabilities (and few competent computer manufacturers will deny the feasibility, even today, of super-speed and interleaved programs) demonstrate a pressing need for an advanced common language system so all users concerned can integrate their particular operations into the complex of control demanded by an automated future.

Just one last prediction—the engineer who is going to be at the top of his profession in the years to come had better become a computer expert, too.

A WORKING GLOSSARY OF SOME AUTOMATIC CODING TERMS

AUTOMATIC CODING — Systems which allow programs to be written in a synthetic language especially designed for problem statement, which the processor translates to presumably the most efficient final machine language code for any given computer. Usually such a system will examine one entry at a time and produce some amount of coding which is determined by that entry alone.

AUTOMATIC PROGRAMMING—Systems further up the scale of complexity, where the computer program helps to plan the solution of the problem as well as supply detailed coding. Such systems usually examine many entries in parallel and produce optimized coding where the result of any single entry depends upon its interactions with other entries.

ASSEMBLER—An original generic name for a processor which converts, on a one-for-one basis, the synthetic language entries to machine instructions. This process occurs prior to the actual execution of the working program. It is a one-level processor which can combine several sections or different programs into an integrated whole, meanwhile assigning actual operation codes and addresses to the instructions.

COMPILER — Generally a more powerful processor than the assembler, although there is a great deal of confusion and overlapping of usage between the two terms. The compiler is capable of replacing single entries with pre-fabricated series of instructions or sub-routines, incorporating them in the program either in-line or in predetermined memory positions with standard mechanisms for entry from and exit to the main routine. Such compound entries are sometimes called "macro-instructions." The basic principle of a compiler is to translate and apply as much intelligence as possible ONCE before the running of the program, to avoid time-consuming repetition during execution. It produces an expanded and translated version of the original, or source program. According to the ACM, a compiler may also produce a secondary synthetic program for interpretation while running.

FLOATING POINT—Number notation whereby a number X is represented by a pair of numbers Y and Z in the form: $X = Y \cdot B^{z}$ where B is the number base used. For floating decimal notation the base B is 10; for floating binary the base is 2. The quantity Y is called the fraction or mantissa, and in the best notation $O = Y - 1$. Z is an integer called the exponent or power.

GENERATOR—A generator is a program which writes other programs, usually on a selective basis from given parameters and skeletal coding. It may be either a character-controlled generator, so that it selects among several options according to a preset character matrix, or a pure generator, which writes a program on the basis of calculations which it makes from the input data. Almost all assemblers and compilers have generating elements in some form.

INDEX REGISTER — A register whose contents are used to automatically modify addresses incorporated in instructions just prior to their execution, the original instruction remaining intact and unmodified in memory. It may either be built in the hardware and circuitry of the computer or be simulated by the program. The original unmodified addresses are termed presumptive, the modified addresses are termed effective.

INTERPRETER—In contrast to an assembler, compiler or generator, a source program designed for interpretation is converted to an object program which is not in machine language when run. The interpreter itself is an executive program which must always be used in conjunction with the object program, and always resides in high-speed memory during execution of the problem. The object program is always subservient to the interpreter, which fabricates from the incompletely converted instruction the necessary parts of machine language instructions just before they are executed. Each entry in the interpretive language usually calls for the use of a complete subroutine, which is used again and again by filling in the missing parts of certain of its instructions from the object instruction.

MACHINE LANGUAGE—The wired-in circuftry language at a low logical level which is intelligible to the computer. It should seldom be used to code problems because of the difficulties of usage at this level and the tendency to error.

OBJECT PROGRAM—The output of the processor when it has translated the source program to either machine language or a second level synthetic language.

PROCESSOR—Also called a translator, this is a computer program which produces other programs, in contrast to programs which are working and produce answers.

SOURCE PROGRAM—The original program written to solve problems and produce answers, phrased in the synthetic language.

PRINT 1—AN AUTOMATIC CODING SYSTEM FOR THE IBM 705

By

ROBERT W. BEMER

FRANKLIN INSTITUTE OF THE STATE OF PENNSYLVANIA. 1824

Reprinted from Monograph No. 3
JOURNAL OF THE FRANKLIN INSTITUTE Series
April, 1957

8

PRINT 1—AN AUTOMATIC CODING SYSTEM FOR THE IBM 705

BY

ROBERT W. BEMER[1]

PURPOSE

PRINT 1 is an automatic coding system for the IBM 705, primarily for use in scientific and engineering applications. It is fully symbolic and provides simulated floating point operation and index registers. It is not to replace or supersede the Autocoder system for business and commercial problems although it has this capability in a more limited form if needed. Both are concurrent products of the Programming Research Department of IBM, differing primarily in emphasis of application. There is no need to fuse the two systems inasmuch as better and more advanced common language systems are presently being developed.

HISTORY

The development of PRINT was essentially an emergency measure to have an engineering computing system for the "705" in operation as soon as possible. For this reason, PRINT is not at the level of the FORTRAN system for the "704" and advantage was not taken of the total automatic coding knowledge available at its conception. Coding was started in February 1956 and the system was being tested by the first customer at the end of July. Because of the interpretive nature of PRINT it was actually completed before Autocoder and FORTRAN. Copies of the completed system were distributed generally with the manuals in October 1956. The responsibility for maintenance and further development of the system now lies with the Applied Programming Department of IBM.

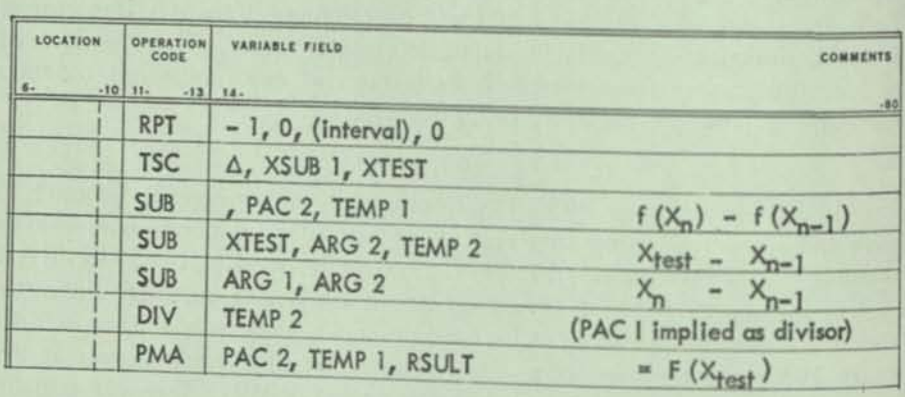
USAGE AND EXPERIENCE

PRINT 1 is in operation in the field and may be considered rather thoroughly tested at this time although, as with the computer itself, continuing maintenance is required to add improvements as they become obvious or are requested by the users. Such changes will not be allowed if they refute the fixed principles of operation or introduce incompatibility. By the last count there are 28 installations either using or programming to use the PRINT system when delivery permits. Many of these have had fairly extensive experience by this time and have given helpful comments and suggestions.

[1] Assistant Manager of Programming Research, International Business Machines Corporation, New York N. Y.

PRINT 1 has given much evidence of the potential of current automatic coding systems. One example is furnished by the A. O. Smith Corporation of Milwaukee, Wisconsin, a long time user of computing equipment. In their initial attempt, a portion of a problem formerly requiring 305 instructions in "705" machine language was recoded using 8 PRINT instructions. With a relatively small staff, the A. O. Smith people feel that it might have been impossible to get into satisfactory operation so soon had it not been for the availability of PRINT 1. I should add that we also owe them (Bob Brittenham in particular) a debt for the many excellent suggestions leading to some of the best instructions in the PRINT repertoire.

Another example was furnished to us by Westinghouse Electric at Sharon, Pennsylvania, where one of their best programmers had previously coded a magnetic field parameter study. This program contained 2300 "705" instructions and required a week to write, a remarkable feat in itself, representing as it does nearly one instruction written per minute. After a single day's instruction in the use of the PRINT 1 system, he recoded the problem in 60 PRINT instructions and took only 20 minutes to do so. Examples like this justify our contention that there may be as much as a 40 to 1 ratio between "705" instructions executed and PRINT instructions written, and in this case the time required to write the program was reduced to less than a hundredth of the time formerly required. Such examples are perhaps exceptional, but it is quite generally true that detailed coding (not programming) effort may be reduced by a factor of 10 through the use of this and other modern automatic coding systems such as FORTRAN, the Carnegie Tech Compiler, BACAIC, B-Zero and OMNICODE.



LOCATION 6- -10	OPERATION CODE 11- -13	VARIABLE FIELD 14-	COMMENTS -80
	RPT	- 1, 0, (interval), 0	
	TSC	Δ, XSUB 1, XTEST	
	SUB	, PAC 2, TEMP 1	$f(X_n) - f(X_{n-1})$
	SUB	XTEST, ARG 2, TEMP 2	$X_{test} - X_{n-1}$
	SUB	ARG 1, ARG 2	$X_n - X_{n-1}$
	DIV	TEMP 2	(PAC I implied as divisor)
	PMA	PAC 2, TEMP 1, RSULT	$= F(X_{test})$

FIG. 1.

Another significant omen arose from the PRINT class at Westinghouse. Two people from National Supply attended, although they had had no previous computer experience at all, and were able to write successful programs in PRINT without having ever learned to program for the "705" itself.

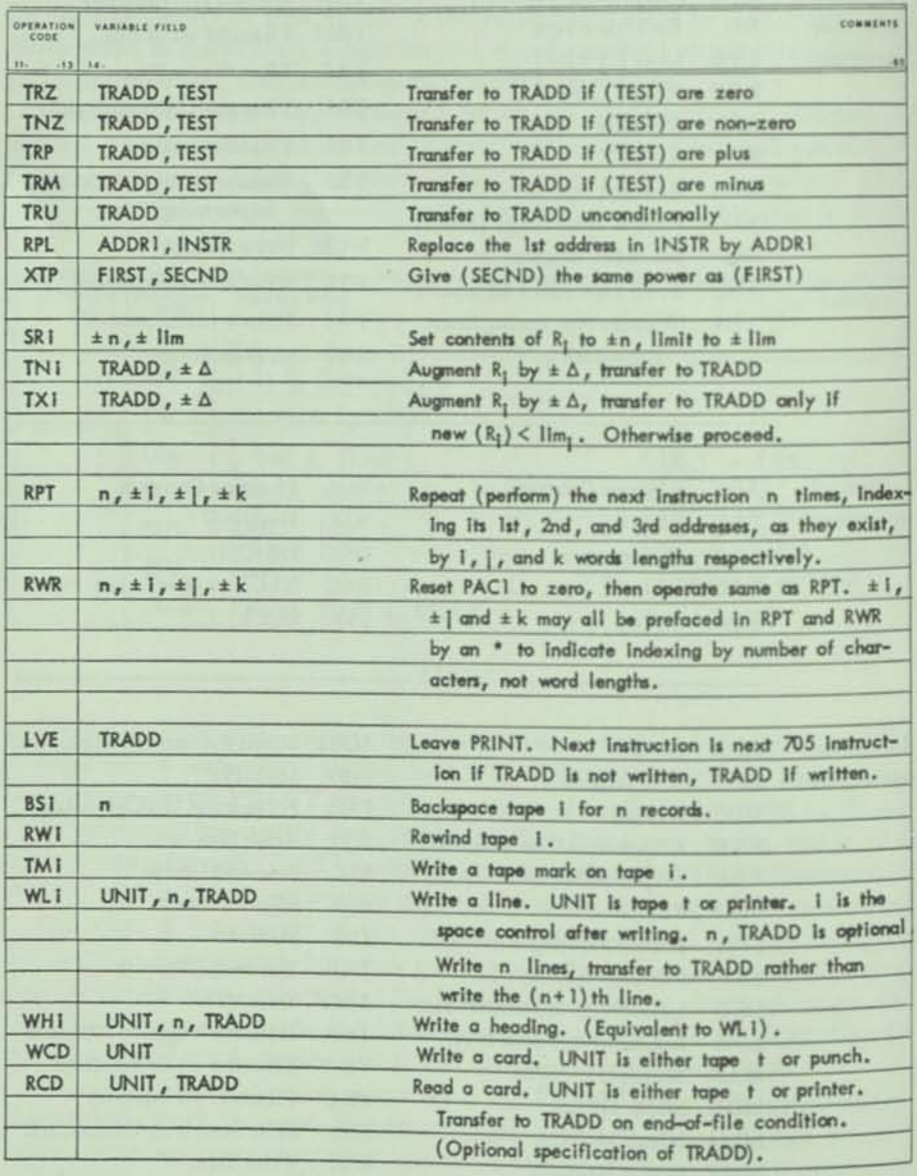
SIMULATED HARDWARE

Since the "705" is not provided with hardware usually considered vital to easy programming of scientific problems, such hardware had to be simulated within the coding system. Floating point arithmetic is furnished in one of three system tapes, for fraction lengths of 8, 10 and 12 digits, with mathematical subroutines to corresponding accuracies. Three index reg-

Non-indexable operations	ATR	Alternating TRansfer	TNZ	Transfer on Non-Zero
	BSi	BackSpace tape "i"	TRM	TRansfer on Minus
	LVE	LeaVE PRINT	TRP	TRansfer on Plus
	RCD	Read a CarD	TRU	TRansfer Unconditionally
	RPL	RePLace	TRZ	TRansfer on Zero
	RPT	RePeaT	TXi	Transfer testing indeX limit, augmenting "i"
	RWi	ReWind tape "i"	WCD	Write a CarD
	RWR	Repeat With Reset (PACI)	WHi	Write a Heading, space "i"
	SRi	Set index Register "i"	WLi	Write a Line, space "i"
	TMi	write Tape Mark on tape "i"	XTP	eXTract Power
	TNi	Transfer Not testing limit, augmenting "i"		
Special operations	ADC	ADdress Constant	FLC	FLoating Constant
	BLK	BLocK	HDG	HeaDinG
	CON	CONstant	ORG	ORiGin
	DEL	DELete	REG	REGister reservation
	FIN	FINish	SAY	SAY it
Indexable operations	ADD	ADD	MPM	Minus Polynomial Mult.–add
	ART	ARcTangent	MPY	MultiPlY
	DIV	DIVide	PMA	Polynomial Multiply–Add
	EXD	EXponential; Decimal base	RTi	Read Tape "i"
	EXE	EXponential, base E (e)	SAC	Sine And Cosine
	FLO	FLOat	SQR	SQuare Root
	FPR	Fix for Printing Rounded	SUB	SUBtract
	FXP	FiX for Printing	TAB	Transmit ABsolute
	LGD	LoGarithm to Decimal base	TMT	TransMiT
	LGE	LoGarithm to base E (e)	TNA	Transmit Negative Absolute
	MAD	Multiply – ADd	TRC	TRansfer on Comparison
	MDV	Minus DiVide	TRE	TRansfer on Equality
	MMA	Minus Multiply – Add	TSC	Table Search on Comparison
	MMY	Minus MultiplY	WTi	Write Tape "i"

FIG. 2. Summary of mnemonic codes.

isters which may be used compositely are furnished, together with corresponding limit registers for incremental loop termination. Memory images for the printed line, heading and card form are symbolically addressable so that the programmer has the feeling of actually addressing type wheel or card column. Other special registers, such as pseudo-accumulators, also maintain invariant symbolic addresses.

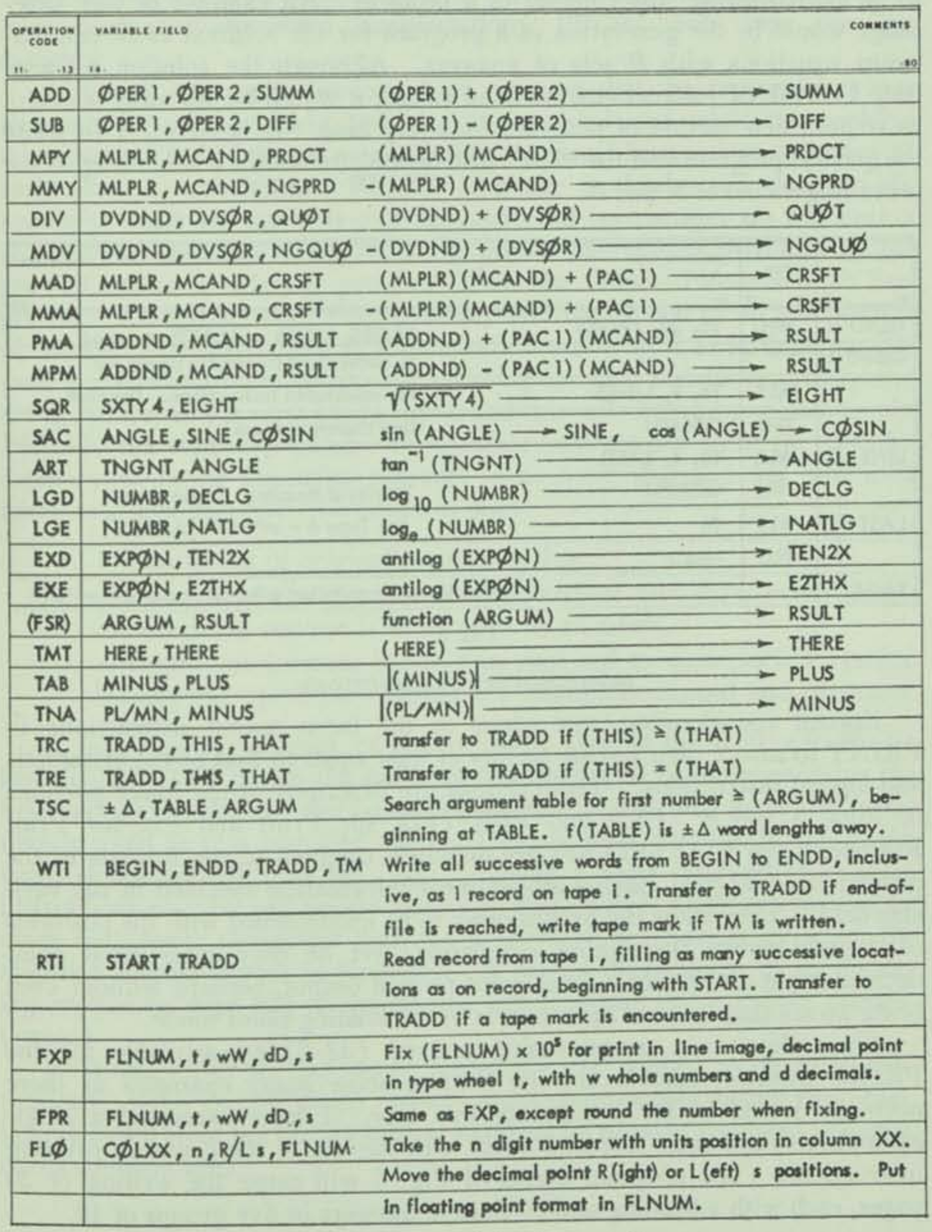


OPERATION CODE 11- -13	VARIABLE FIELD 14-	COMMENTS
TRZ	TRADD, TEST	Transfer to TRADD if (TEST) are zero
TNZ	TRADD, TEST	Transfer to TRADD if (TEST) are non-zero
TRP	TRADD, TEST	Transfer to TRADD if (TEST) are plus
TRM	TRADD, TEST	Transfer to TRADD if (TEST) are minus
TRU	TRADD	Transfer to TRADD unconditionally
RPL	ADDR1, INSTR	Replace the 1st address in INSTR by ADDR1
XTP	FIRST, SECND	Give (SECND) the same power as (FIRST)
SRI	± n, ± lim	Set contents of R_i to ±n, limit to ± lim
TNI	TRADD, ± Δ	Augment R_i by ± Δ, transfer to TRADD
TXI	TRADD, ± Δ	Augment R_i by ± Δ, transfer to TRADD only if
		new $(R_i) < \text{lim}_i$. Otherwise proceed.
RPT	n, ± i, ± j, ± k	Repeat (perform) the next instruction n times, index-
		ing its 1st, 2nd, and 3rd addresses, as they exist,
		by i, j, and k words lengths respectively.
RWR	n, ± i, ± j, ± k	Reset PAC1 to zero, then operate same as RPT. ± i,
		± j and ± k may all be prefaced in RPT and RWR
		by an * to indicate indexing by number of char-
		acters, not word lengths.
LVE	TRADD	Leave PRINT. Next instruction is next 705 instruct-
		ion if TRADD is not written, TRADD if written.
BSi	n	Backspace tape i for n records.
RWi		Rewind tape i.
TMi		Write a tape mark on tape i.
WLi	UNIT, n, TRADD	Write a line. UNIT is tape t or printer. i is the
		space control after writing. n, TRADD is optional
		Write n lines, transfer to TRADD rather than
		write the (n+1)th line.
WHi	UNIT, n, TRADD	Write a heading. (Equivalent to WLi).
WCD	UNIT	Write a card. UNIT is either tape t or punch.
RCD	UNIT, TRADD	Read a card. UNIT is either tape t or printer.
		Transfer to TRADD on end-of-file condition.
		(Optional specification of TRADD).

FIG. 3. Summary of non-indexable operations.

INSTRUCTIONS

PRINT instructions are written in a variable form with a variable number of operands or specifications separated by commas. A single instruction may trigger several actions which are effectively coincident. A coding kernel for table search and linear interpolation is shown in Fig. 1 (refer to Figs. 2, 3, 4 for explanation of the instructions). The table search



OPERATION CODE	VARIABLE FIELD	COMMENTS
11 13	14	80
ADD	ØPER 1, ØPER 2, SUMM	(ØPER 1) + (ØPER 2) → SUMM
SUB	ØPER 1, ØPER 2, DIFF	(ØPER 1) - (ØPER 2) → DIFF
MPY	MLPLR, MCAND, PRDCT	(MLPLR) (MCAND) → PRDCT
MMY	MLPLR, MCAND, NGPRD	-(MLPLR) (MCAND) → NGPRD
DIV	DVDND, DVSØR, QUØT	(DVDND) + (DVSØR) → QUØT
MDV	DVDND, DVSØR, NGQUØ	-(DVDND) + (DVSØR) → NGQUØ
MAD	MLPLR, MCAND, CRSFT	(MLPLR) (MCAND) + (PAC 1) → CRSFT
MMA	MLPLR, MCAND, CRSFT	-(MLPLR) (MCAND) + (PAC 1) → CRSFT
PMA	ADDND, MCAND, RSULT	(ADDND) + (PAC 1) (MCAND) → RSULT
MPM	ADDND, MCAND, RSULT	(ADDND) - (PAC 1) (MCAND) → RSULT
SQR	SXTY 4, EIGHT	$\sqrt{\text{(SXTY 4)}}$ → EIGHT
SAC	ANGLE, SINE, CØSIN	sin (ANGLE) → SINE, cos (ANGLE) → CØSIN
ART	TNGNT, ANGLE	$\tan^{-1}$ (TNGNT) → ANGLE
LGD	NUMBR, DECLG	$\log_{10}$ (NUMBR) → DECLG
LGE	NUMBR, NATLG	$\log_e$ (NUMBR) → NATLG
EXD	EXPØN, TEN2X	antilog (EXPØN) → TEN2X
EXE	EXPØN, E2THX	antilog (EXPØN) → E2THX
(FSR)	ARGUM, RSULT	function (ARGUM) → RSULT
TMT	HERE, THERE	(HERE) → THERE
TAB	MINUS, PLUS	|(MINUS)| → PLUS
TNA	PL/MN, MINUS	|(PL/MN)| → MINUS
TRC	TRADD, THIS, THAT	Transfer to TRADD if (THIS) ≥ (THAT)
TRE	TRADD, THIS, THAT	Transfer to TRADD if (THIS) = (THAT)
TSC	± Δ, TABLE, ARGUM	Search argument table for first number ≥ (ARGUM), be-
		ginning at TABLE. f(TABLE) is ± Δ word lengths away.
WTI	BEGIN, ENDD, TRADD, TM	Write all successive words from BEGIN to ENDD, inclus-
		ive, as 1 record on tape I. Transfer to TRADD if end-of-
		file is reached, write tape mark if TM is written.
RTI	START, TRADD	Read record from tape I, filling as many successive locat-
		ions as on record, beginning with START. Transfer to
		TRADD if a tape mark is encountered.
FXP	FLNUM, t, wW, dD, s	Fix (FLNUM) x 10^s for print in line image, decimal point
		in type wheel t, with w whole numbers and d decimals.
FPR	FLNUM, t, wW, dD, s	Same as FXP, except round the number when fixing.
FLØ	CØLXX, n, R/L s, FLNUM	Take the n digit number with units position in column XX.
		Move the decimal point R(ight) or L(eft) s positions. Put
		in floating point format in FLNUM.

FIG. 4. Summary of indexable operations.

is initiated by the first two instructions. Both a coarse and a fine search are caused, and on completion both the bracketing table arguments and their corresponding functions may be found in special locations.

Although it is more desirable to code throughout in PRINT, it is possible to exit and code in symbolic "705" language temporarily. This allows both for the writing of special portions of programs and for PRINT instructions to be altered during the running of a problem. An example of this latter usage would be the generation of a program for the solution of *N* simultaneous equations with *B* sets of answers. Although the solution requires only 11 PRINT instructions for fixed values, it would be normally wasteful to write a new section of program for every case. It is sufficient to write the general program and use "705" instructions to modify it according to the prevailing values of *N* and *B*.

LOCATION		OPERATION CODE	VARIABLE FIELD	COMMENTS
6-	-10	11- -13	14-	-80
HEAD		WHT	T6, 20, PAGES	PAGES, LINES and LASTL are used
CØMP	U	— — —	- - - -	as convenient mnemonic names for
		WLS	T6, 9, LINES	the associated instructions. The first
		TRU	CØMPU	line therefore reads:
LINE	S	WLD	T6, 4, LASTL	
		TRU	CØMPU	"Write a Heading, Triple space,
LAST	L	WLI	T6	on Tape 6 - write 20 PAGES."
		TRU	HEAD	
PAGE	S			(continues computation after 20 pages are written)

FIG. 5.

INPUT–OUTPUT INSTRUCTIONS

Special consideration was given to the input–output instructions in PRINT to assure their having facility at least equal to that of the arithmetic and logical instructions. Their actions are described in the lower portions of Figs. 3 and 4. FXP and FPR (FiX for Print and Fix for Print Rounded) and FLO (FLOat) are definitely oriented to the formats of the printed line and card, having all pertinent information specified in the variable field. They allow the programmer to be unconcerned with the positions of decimal points throughout calculation; yet he may enter fixed point decimal input and produce fixed point printed output, perhaps without even being aware that internal operation is in the floating point mode.

Since there is a programming manual (32–7334) available for the PRINT system, there is not the need to show many examples as there normally would be for a paper of this type. I have excerpted a single example to show page format control with the use of the counting printing instructions. The program kernel of Fig. 5 will cause the writing of 20 pages, each with a heading and 50 lines of answers in five groups of 10.

INTERNAL OPERATION

Certain definitions are adopted from the 704 FORTRAN system in order to understand the hybrid operation of the PRINT system. A program written in the synthetic automatic coding language is called a "source" program. It is processed by a translator to produce an "object" program, which may be produced in either a machine language form, a still symbolic intermediate language such as that of an assembly program, or pseudo-instructions for minimum interpretation. PRINT falls into the last category.

Although interpretive in execution, meaning that the required machine language instructions have certain portions fabricated while the problem is running, PRINT is not equivalent to the usual interpretive program of early days in computers. PRINT language is freely and descriptively symbolic, much the same as any compiler, and instructions do not bear a recognizable resemblance to the object pseudo-instructions produced by the pre-editing, or translating, process. Thus the name—PRe-edited INTerpretive. Pre-editing does both assembly and conversion of all components of the synthetic instruction to a pseudo-instruction in a form most rapidly used at execution time, essentially following the first compiling principle of doing all repetitive processing once and for all wherever possible. In PRINT, the time expenditure to fabricate instructions from the pseudo-components during execution amounts to no more than a 5 per cent total addition. For this price the program buys:

1. Minimization of original processing time.
2. Much more memory space for instructions and data, even though the executive routine is in memory at all times.
3. A significant decrease in the time required to write such a system, because the operative routines are essentially canned and optimum.

A factor in the decrease of interpretation time is the RPT (RePeaT) instruction. This causes the following instruction to be interpreted for the first execution only; for the remaining times it is executed generally faster than it could be in a compiled form. This apparent paradox is due to the serial character nature of the "705." Using a fixed interpretive routine, instructions may be judiciously placed so that address modification may be made with fewer characters than the four which are mandatory when the modified address cannot be predicted. A careful examination of the indexing routine on page 52g of the manual will illustrate this principle unquestionably.

Routines which do not occur frequently are defined as floating subroutines. They do not occupy memory space continuously during execution, but are called from the library tape as required by pre-edit compiled linkages, into a common area.

EXTERNAL OPERATION

Because in this semi-interpretive mode the routines are what one might call canned or pre-compiled, there are other advantages which are not at first apparent. It takes very little time to pre-edit the symbolic source program into an object program ready to execute. Very few programs have taken over one minute to process from tape input. Because of this it was decided to re-process the program completely each time a change is made; in fact, it is impossible to correct or alter a PRINT program in anything other than the synthetic language. Any other means were carefully, as Sam Goldwyn says, "included out."

This feature also leads to a radically different concept of error diagnosis. Since the program may be re-processed quickly and flexibly, the most desirable diagnostic method is to actually insert snapshot instructions into the program. This gives the programmer absolute freedom and flexibility in inspecting intermediate answers. The usual method is to process the program initially with all of the snapshot instructions included, when the cause of the first error becomes apparent, to remove all snapshot instructions up to that point by deletion cards and correct the error all in the same reprocessing. Of course, programs are never loaded more than once in card form. All corrections thereafter are made by collating the change or deletion cards against a master symbolic tape. Both this and a listing tape are thus continuously updated to provide a correct, permanent record of the programming of any problem.

This quick processing feature has demanded at least a primitive type of supervisory control directed by the system tape, which contains the pre-edit routine, the executive routine, the floating subroutine library, diagnostics and system control. A remarkably rapid interchange of problems has been achieved. With tapes previously mounted off-line, all that is required to process a new problem is to turn the tapes on-line, set alteration switches if required and depress the reset and start buttons. For the moment, the changing nature of scientific problems has not made a completely supervisory control mandatory for this system.

CONCLUSION

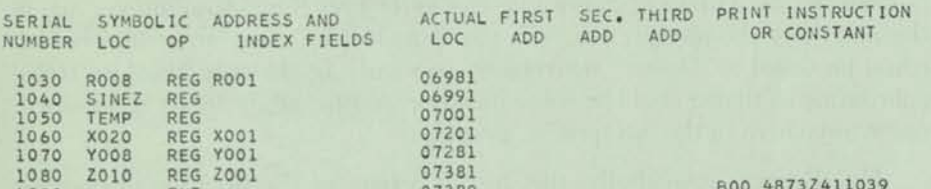
PRINT 1 is an operative scientific computing system for the IBM 705 which allows it to be used by both the commercial and engineering divisions of a company. By introducing pre-editing to take advantage of the lessons learned from compilers, it re-establishes the interpretive method as a useful tool in automatic coding systems for future computers of the STRETCH class. A programming manual (32–7334) is available from IBM for those desiring further information, and the entire system is available on cards upon request.

DISCUSSION

MR. BARRY GORDON[1]: You made quite a point of emphasizing that it was a remarkable thing that the man wrote a "705" program at the rate of one instruction per minute; however, you made very little of his writing sixty PRINT instructions in 20 minutes—at a rate of three instructions per minute. I am wondering how usual this is.

MR. BEMER: I would say that in my estimation it is much easier to write the PRINT instructions because you don't have to worry about the auxiliary storage unit and all the actual details of operation. Incidentally, if you should doubt those figures I gave, let's bump it up to at least the same ratio—this is still 40 to 1 over the original time.

MR. LEROY D. KRIDER[2]: You said something about a compilation that actually went on during the execution?



```
SERIAL  SYMBOLIC   ADDRESS AND      ACTUAL FIRST  SEC. THIRD  PRINT INSTRUCTION
NUMBER  LOC    OP     INDEX FIELDS   LOC    ADD   ADD   ADD       OR CONSTANT

 1030   R008   REG R001              06981
 1040   SINEZ  REG                   06991
 1050   TEMP   REG                   07001
 1060   X020   REG X001              07201
 1070   Y008   REG Y001              07281
 1080   Z010   REG Z001              07381
 1090          ENT                   07389                     B00 4873Z411039
 1100          RCD READER            07400                     7I7DA1100Y4
 1110          RPT 20,*4,1           07411                     2I11000400100000
 1120          FLO COL04,4,L2,X001   07428  03512 07011        9M351200070020D0F
 1130          RCD READER            07445                     7I7DE6100Y4
 1140          RPT 8,*5,1            07456                     2I0G000500100000
 1150          FLO COL05,5,L3,Y001   07473  03513 07211        9M351300072020E0E
 1160          RPT 10,*4,1           07490                     2I0I000400100000
 1170          FLO COL44,4,L3,Z001   07507  03552 07291        9M355200072820D0E
 1180          SR3 0,10              07524                     9DI900 0000
 1190   PAGE   WHT TAPE4             07535                     6I0++00-0+20M38G40
 1200          SR1 0,20              07553                     8DI800 0000
 1210          SAC Z001,3,SINEZ      07564  07291 06991 00244  2R7K8240069820235
 1220          TRU RSETY             07581  07605              2D7F+5
 1230   LINE   WLS TAPE4             07587                     6I0++00-0+20D26N4
 1240   RSETY  SR2 0,8               07605                     8I1920 0000
 1250          SUB X001,1,SINEZ,TEMP 07616  07011 06991 07001  0M7-021006R826992Q
 1260   COLMN  SAC Y001,2            07634  07211 00254 00244  2R7K0220002450235
 1270          ADD X001,1,PAC2       07651  07011 00244 00254  0M7-021000K350245H
 1280          MPY ,TEMP,R001,2      07669  00254 07001 06911  0R0K450026R926902H
 1290          TX2 COLMN,1           07687  07634              4D7FC420010
 1300          RPT 8,1,*11           07698                     2I0G001000110000
 1310          FPR R001,8,4W,2D      07715  06911              9R6R020+00+27C0C0J
 1320          TX1 LINE,1            07733  07587              3I7EH720010
 1330          WL1 TAPE4             07744                     6I0++00-0+20D26N41
 1340          TX3 PAGE,1            07762  07535              4I7EC520010
 1350          LVE                   07773  07784              9117784
```

FIG. A.

[1] Chief Programmer, Equitable Life Assurance Society, New York, N. Y.

[2] Remington Rand Univac, Minneapolis, Minn.

Mr. Bemer: No, there is no true compilation during execution. What there is is the completion of certain machine instructions with portions of the pseudo-instructions. If you have one of the example sheets (Fig. A), you will notice the pseudo-instruction on the right. This "garbage" is really several little portions that are required to fill in empty spots in any routine. For instance, in the instruction with serial number 1130, we see a 7I7DE6100Y4. 7I is the "Read Card" operation code, 7DE6 the properly zoned address for the next instruction, 100 the last three digits of the card reader address, Y for a "705" read-operation code and 4 to fill in a TRS command in the units position. These components are inserted in the proper places of the canned subroutines just before execution, but they are properly detailed with everything that can be done once, so only these portions are inserted during operation.

Mr. Krider: It is essentially, then, just a stretch of initialization.

Mr. Bemer: Yes, it is really interpretive, with the exception that we have put in the symbolic notation and pre-edited everything we possibly could, to speed things up.

Mr. R. H. Doyle[3]: Since one can enter and leave absolute coding in the middle of the interpretive, is it also possible, by using something which could be called a "Define" instruction, to name-tag the new absolute coded subroutine so that it could be subsequently called for again by its interpretive name anywhere in the interpretive program?

Mr. Bemer: Essentially, this facility exists in the floating subroutine. If you wish to enter any of the subroutine library you can do it. Other routines have been used this way by means of a dummy instruction, or actually a pair of instructions. I believe that the description of this operation has been published, or is going to be out shortly, as a customer contribution. It is very definitely possible.

Dr. Hans K. Flesch[4]: I heard you remark on time-sharing the debugging time for programs coded in PRINT. Is that all under system control?

Mr. Bemer: Yes, it is. Processing the synthetic language, running the problem and diagnostic action are all controlled by the system tape, six alteration switches and the reset and start keys.

Moderator Walter F. Bauer[5]: I am a little disappointed that nobody asked him what Machine X and Machine Y are.

Mr. Bemer: They wouldn't have gotten an answer if they had.

[3] International Business Machines Corporation, Lexington, Mass.

[4] Project Engineer, Federal Telecommunications Laboratories, Nutley, N. J.

[5] Head of Digital Computing Center, Ramo-Wooldridge Corporation, Los Angeles, Calif.

RBemer

9

PROCEEDINGS

of the

FOURTH ANNUAL

COMPUTER APPLICATIONS SYMPOSIUM

OCTOBER 24–25, 1957

Sponsored by

ARMOUR RESEARCH FOUNDATION
OF ILLINOIS INSTITUTE OF TECHNOLOGY
Technology Center
Chicago, Illinois

THE STATUS OF AUTOMATIC PROGRAMMING FOR SCIENTIFIC PROBLEMS

R. W. BEMER
International Business Machines Corporation

PURPOSE

This paper is to be something of a "state of the art" survey in the field of automatic coding systems for scientific work. In a rapidly changing field where thc communication processes are by no means satisfactory, it máy be valuable to collect and codify available information on the various systems in use or in process of fabrication, if only to provide a solid basis for dispelling rumor and misconception. Very little new or inventive material will be presented, inasmuch as this does not seem to be the natural means of development in this profession. Development is rather by distillation and blending of certain principles which force themselves upon us as exceptions to the general case, while using the older systems in actual practice. Let us rather, for the moment, delineate trends and put existing efforts into historical perspective, making a summary of the various efforts so far with respect to magnitude and usefulness.

There will undoubtedly be some gaps in the record due to incomplete information. In many cases this is not from negligence on my part but from lack of proper publicity and communication (publicity departments, please note). I further hope that this is one of the very last mentions of scientific computing as a separate entity; the very near future will bring us new systems that encompass both business and scientific applications and allow each group of users to have the powers formerly peculiar to the other.

EXISTING CODING SYSTEMS

Since most automatic coding systems have heretofore had their synthetic languages tied closely to the particular computer on which they are used, the easiest way to make a résumé of these is by machine categories. Table 1 is an updated version of a chart which previously appeared in the March, 1957, issue of *Automatic Control Magazine.* Two copies each of all available information and manuals on these systems are being deposited with the Association for Computing Machinery at 2 East Sixty-third Street, New York City. It is my hope that the ACM will see fit either to lend material from this historical library or to refer interested correspondents to the proper sources for other copies.

The more important or widely used scientific systems have a dagger preceding the name. (Note that B-ZERO [or FLOWMATIC] is listed as scientific because it accepts AT3 as well as other language.) Since there are ninety systems cata-

TABLE 1*

NBS CORBIE

OPERATIONAL AUTOMATIC CODING SYSTEMS FOR SCIENTIFIC PROBLEMS

Computer	Name or Acronym of Automatic Coding System	Developed By	Operational Date	Use Code	Interpreter	Assembler	Compiler	Machine Lang.	Symbolism	Algebraic	Indexing	Floating Pt.
IBM 704	†AFAC	Allison GM	Sep 57	C			x		2	x	M2	M
	CAGE ET-MO	General Electric	Nov 55			x	x		2		M2	M
	†FORC H.SASSENFELD	Redstone Arsenal	Jun 57				x		2	x	M2	M
	†FORTRAN	IBM	Jan 57	A			x		2	x	M2	M
	NYAP	IBM	Jan 56			x			2		M2	M
	†PACT IA	PACT Group	Jan 57				x		1		M2	M
	REG.-SYMBOLIC	Los Alamos	Nov 55			x			1		M2	M
	SAP	United Aircraft	Apr 56	R		x			2		M2	M
IBM 701	ACOM	Allison GM	Dec 54	C	x				0		S1	S
	†BACAIC	Boeing Seattle	Jul 55	A		x	x		1	x		S
	DOUGLAS	Douglas (SM)	May 53			x			1			S
	DUAL	Los Alamos	Mar 53		x			x	1			S
	607	Los Alamos	Sep 53			x			1			
	FLOP	Lockheed Calif.	Mar 53		x	x		x	1			S
	JCS 13	RAND Corp.	Dec 53			x			1			
	KOMPILER 2	UCRL Livermore	Oct 55				x		1	x	S2	
	NAA ASSEMBLY	North American			x							
	†PACT I	PACT Group	Jun 55	R			x		1		S2	
	QUEASY	NOTS Inyokern	Jan 55		x							S
	QUICK	Douglas (ES)	Jun 53		x				0			S
	SHACO	Los Alamos	Apr 53		x				1			S
	SO 2 DI GRI	IBM	Apr 53			x			1			
	SPEEDCODING	IBM	Apr 53	R	x	x			1		S1	S
IBM 705-1, 2	ACOM	Allison GM	Apr 57	C	x				0		S1	
	AUTOCODER	IBM	Dec 56	R		x	x	x	2			S
	ELI	Equitable Life	May 57	C	x				0		S1	
	FAIR	Eastman Kodak	Jan 57		x				0			S
	†PRINT I	IBM	Oct 56	R	x	x		x	2		S2	S
	SYMB. ASSEMBLY	IBM	Jan 56			x			1			
	SOHIO	Std. Oil of Ohio	May 56		x	x		x	1		S1	S
IBM 702	AUTOCODER	IBM	Apr 55			x	x	x	1			S
	ASSEMBLY	IBM	Jun 54			x			1			
	†SCRIPT	GE Hanford	Jul 55	R	x	x	x	x	1		S1	S
IBM 650	†ADES II	Naval Ordnance Lab.	Feb 56				x		1	x	S2	S
	†BACAIC	Boeing Seattle	Aug 56	C	x	x	x		1	x		S
	BALITAC	MIT	Jan 56			x	x	x	2		S1	
	†BELL L1	Bell Tel. Labs.	Aug 55		x			x	0		S1	S
	BELL L2, L3	Bell Tel. Labs.	Sep 55		x			x	0		S1	S
	DRUCO I	IBM	Sep 54		x				0			S
	EASE II A.WEISS	Allison GM	Sep 56		x	x			2		S2	S
	ELI GORDON	Equitable Life	May 57	C	x				0		S1	
	ESCAPE HORNER	Curtiss-Wright	Jan 57		x	x	x		2		S1	S
	FLAIR	Lockheed MSD. Ga.	Feb 55		x			x	0		S1	S
	†FOR TRANSIT	IBM–Carnegie Tech	Oct 57	A			x		2	x	S2	S
	†IT	Carnegie Tech.	Feb 57	C			x		1	x	S2	S -D.P.
	MITILAC	MIT	Jul 55		x			x	2		S1	S
	OMNICODE	GE Hanford	Dec 56		x		x		2		S1	S
	RELATIVE HORNER	Allison GM	Aug 55			x			1		S1	S
	SIR	IBM	May 56		x				2			S
	SOAP I	IBM	Nov 55			x			2			
	SOAP II	IBM	Nov 56	R		x			2		M	M
	SPEED CODING	Redstone Arsenal	Sep 55		x			x	0		S1	S
	SPUR	Boeing Wichita	Aug 56		x	x		x	1		M	S

* See explanation of symbols at end of Table 2.

TABLE 1—*Continued*

Computer	Name or Acronym of Automatic Coding System	Developed By	Operational Date	Use Code	Interpreter	Assembler	Compiler	Machine Lang.	Symbolism	Algebraic	Indexing	Floating Pt.
Sperry Rand 1103A	†COMPILER I	Boeing Seattle	Mar 57			x	x		1	x		S
	FAP MODIFIED	Lockheed MSD	Oct 56		x			x	0		S1	S
	MISHAP RAWOOP	Lockheed MSD	Oct 56			x			1		M1	S
	RAWOOP-SNAP	Ramo-Wooldridge	Jun 57		x	x			1		M1	M
	TRANS-USE	Holloman AFB	Nov 56			x			2		M1	S
	†USE	Ramo-Wooldridge	Feb 57	R		x	x		2		(M1)	(M)
Sperry Rand 1103	CHIP	Wright ADC	Feb 56		x			x	0		S1	S
	FLIP/SPUR	Convair San Diego	Jun 55		x			x	0		S1	S
	RAWOOP	Ramo-Wooldridge	Mar 55	R		x			1		S1	
	SNAP	Ramo-Wooldridge	Aug 55	R	x	x			1		S1	S
Sperry Rand Univac I, II	A0 (A-ZERO)	Sperry Rand	May 52			x	x	x	1		S1	S
	A1	Sperry Rand	Jan 53			x	x	x	1		S1	S
	A2	Sperry Rand	Aug 53			x	x	x	1		S1	S
	†A3	Sperry Rand	Apr 56	C		x	x	x	1		S1	S
	†AT3	Sperry Rand	Jun 56	C		x	x		2	x	S1	S
	†B0 (FLOWMATIC)	Sperry Rand	Dec 56	A		x	x	x	2		S2	S
	BIOR	Sperry Rand	Apr 55			x	x	x	1			
	GP	Sperry Rand	Jan 57	R		x	x	x	1		S2	S
	MJS (I only)	UCRL Livermore	Jun 56			x		x	1			
	NYU (OMNIFAX)	New York Univ.	Feb 54				x		1			S
	RELCODE	Sperry Rand	Apr 56			x		x	1			
	SHORT CODE	Sperry Rand	Feb 51		x			x	1			S
	X-1	Sperry Rand	Jan 56	C		x		x	1			
Datatron 205	DATACODE I ROSEN	Burroughs LIKE A2	Aug 57				x		1		M S1	Ⓢ M 3-ADDR, 2 OR 1 ADDR, SAME PSEUDO
	DUMBO M. SPEISTMAN	Babcock & Wilcox LYNCHBURG, VA.			x OR		x					
	†IT ORGEL NO ALPHA	Purdue Univ.	Jul 57	A			x		1	x	S2	S
	SAC [illegible]	Electrodata LIKE GP	Aug 56			x		x	1			M
	UGLIAC	United Gas Corp.	Dec 56		x				0			S
Whirlwind	ALGEBRAIC	MIT		R			x		1	x	S2	S
	COMPREHENSIVE	MIT	Nov 52		x	x		x	1		S1	S
	SUMMER SESSION	MIT	Jun 53		x				1		S1	S
Midac	EASIAC	Univ. of Michigan	Aug 54		x	x			1		S1	S
	MAGIC	Univ. of Michigan	Jan 54 MAR 55			x	x	x	1		S1 M1	S
Ferranti Ferut	TRANSCODE	Univ. of Toronto	Aug 54	R	x	x	x		1		M1	S
Illiac	DECIMAL INPUT	Univ. of Illinois	Sep 52	R		x			1		S1	S
Johnniac	EASY FOX	RAND Corp.	Oct 55	R		x			1			S
Norc		Naval Ordnance Lab.	Feb 56			x	x		1		M2	M
Seac	BASE 00	Nat'l Bureau Stds.			x			x				
SWAC												

logued, at least seventy of which are suitable for scientific usage, it will be impossible to give detailed information on more than a selected few.

IBM 704.—By far the most ambitious and widely used of these working systems is FORTRAN for the 704 (F4). Several reports have already been given of this system—at the 1957 Western Joint Computer Conference, the 1957 ACM Conference, and the September, 1957, SHARE meeting at San Diego—so it would be redundant to go into any detail here. However, an over-all summary of experience would show that the use of FORTRAN reduces both coding costs and elapsed time of coding by an average factor of five, according to careful statistics, particularly those of the New York City Service Bureau, which bids for programming jobs on this basis. There is an obvious corollary to this which is not always made explicit: since programming costs have been running at least equal to the other installation costs (including rental, overhead, and physical plant), it could be said that an installation using FORTRAN exclusively can operate at 60 per cent of the previous cost.

The FORTRAN system has firmly established the continuous statement in algebraic form as a practical input and has demonstrated conclusively that a running program can be optimized by Monte Carlo and other statistical techniques to a point where it is almost always competitive with the tight coding of the most experienced programmers. A revised version of FORTRAN for the 704, known as F4-2, is currently in process. Major improvements are (1) the addition of a facility to write and name subroutines in either FORTRAN or SAP machine language, compile and optimize locally only into relative binary packages for conditional inclusion, and (2) improved diagnostic facilities. This version is due for release in the spring of 1958.

The other 704 system in extensive use is PACT IA, used particularly on the West Coast by the original co-operative group that did the coding. It is described in detail in the *Journal* of the ACM for October, 1956. It is of particular historical interest because it was the first major co-operative coding effort, and many people believe that this technique will be our only solution when automatic coding gets many times more complex than it is now.

IBM 650—DATATRON.—The IT system, started by Dr. Alan Perlis for the DATATRON while at Purdue and later completed for the 650 at Carnegie Institute of Technology, is the only full-scale scientific system for either the 650 or the DATATRON. IT (Internal Translator) is used extensively at many university computing laboratories and scientific installations. The output of this system is in a form which is further processed into a machine-language object program by the SOAP assembly program for the 650. IBM has just completed final testing of a superstructure to IT, called FOR TRANSIT, which translates a subset of the FORTRAN language to IT language in an initial processing. This is a significant result because it demonstrates how higher-level synthetic languages may be made compatible through pre-processors. IT is also nearly completed for the DATATRON 205, except that the alpha-numeric input-output routines are not complete, since Purdue does not have this type of equipment yet, as I am given to understand.

PACT II →
LARC GP H&T-
G-15A BENDIX-HUSKEY
DTRON 220 DATACODE, IT Hop20

TABLE 2

x x

Automatic Coding Systems in Process of Development

Computer	Name or Acronym of Automatic Coding System	Being Developed By	Expected Operational Date	Use Code	Interpreter	Assembler	Compiler	Machine Lang.	Symbolism	Algebraic	Indexing	Floating Pt.
IBM Tape 650	FORTRAN	IBM ✓		A			x		2	x	M2	M
IBM 709	FORTRAN	IBM ✓	Aug 58	A			x		2	x	M2	M
IBM 705-1,2	FORTRAN	IBM-GUIDE ✓	Aug 58	A			x		2	x	S2	S
IBM 705-3	FORTRAN	IBM-GUIDE ✓	Dec 58	A			x		2	x	M2	M
IBM 705	IT RAY HITT	Std. Oil of Ohio		C			x		1	x	S2	S
Univac	IT F. WAY	Case Institute ✓	JUL 58	C			x		1	x	S2	S
1103A	IT	Carnegie Tech–Ramo-Woold. ✓	Dec 57	C			x		1	x	S2	S
1103A	UNICODE FORTRAN DESILETS	Sp. Rand, St. Paul	Oct 58				x		2	x		
	APS GOTTMAN, OSTROWSKY	Westinghouse Res, PITTS								x		
IBM 709	SCAT	IBM-SHARE	Mar 58			x	x		2		M2	M
IBM 705-3	AUTOCODER †	IBM ✓	Sep 58	R		x	x	x	2			S
IBM 704	NYDPP	Service Bureau Corp. ✓	Sep 57									
IBM 705-1,2	AFAC	Allison GM ✓		C			x		2	x	S2	S
IBM 704	KOMPILER 3	UCRL Livermore					x		2	x	M2	M
File-Computer	ABC KRIDER	Sp. Rand, St. Paul	Jun 58									
Datamatic	ABC I GAUDETTE	Datamatic ✓					x					
Udec III	UDECIN 1 BLOCK	Burroughs ✓	Dec 57		x				1		M/S	S
Udec III	UDECOM 3 T30	Burroughs ✓	Dec 57				x		1		M	S
Datatron 205	FRANK LANDEE	Dow Chemical ✓	AUG 57				X		0		M-1, S-M	
Datatron 205, 220	STAR JIM SIMONE	Burroughs ✓		C		x			½			
Univac	MATRIX MATH	Franklin Inst. ✓	Jan 58 manual				x					
TX-2		Lincoln Labs. ✓										
Stretch	FORTRAN	IBM ✓		R			x			x		
EDVAC, ORDVAC	UNIVERSAL CODE GORN	Moore School ✓	Apr 55				x					

No I/O YET, V.

2 F.DOC

EXPLANATION OF SYMBOLS, TABLES 1 AND 2

Automatic Coding System Name: Dagger (†) indicates more important or widely used scientific system.

Developed By: The PACT Group contains Douglas (Santa Monica, El Segundo, Long Beach), Lockheed, NOTS Inyokern, North American, and RAND Corp.

Use Code:

R Recommended for this computer, sometimes only for heavy usage.
C Common language for more than one computer.
A System is both recommended and has common language.

Machine Lang.: User has option of using machine language together with synthetic.

Symbolism:

0 None.
1 Limited, either regional, relative, or exactly computable.
2 Fully descriptive English word, or symbol combination which is descriptive of the variable or the assigned storage.

Algebraic: A single continuous algebraic formula statement may be made. Processor has mechanisms for applying associative and commutative laws to form operative program.

Indexing:

M Actual index registers or B-boxes in machine hardware.
S Index registers simulated in synthetic language of system.
1 Limited form of indexing, either stepped unidirectionally or by one word only, or having certain registers applicable to only certain variables, or not compound (by combination of contents of registers).
2 General form; any variable may be indexed by any one or a combination of registers which may be freely incremented or decremented by any amount.

Floating Pt.:

M Inherent in machine hardware.
S Simulated in language.

1103A.—There are two existing scientific compilers for this machine: the COMPILER I of Boeing Seattle, and USE, the official compiler of that organization, prepared by Ramo-Wooldridge and others. Although both have many attractive features, they are not quite equal to FORTRAN in power, for USE does not use the algebraic format, and COMPILER I, which does, does not allow subscripting in the algebraic statement.

UNIVAC.—AT3 is an algebraic coding system for UNIVAC I and II which closely resembles FORTRAN. (I should mention here that, when I compare another system to FORTRAN, I mean to establish only the category that system fits into and not to evaluate it against FORTRAN, for there are many features of AT3 and other languages which are perhaps improvements on the corresponding FORTRAN components.) AT3 has been used for a year, with a preliminary manual, in two installations and is about to be released with its final manual for general usage. For purposes of negotiability of programs, if desired, AT3 is mappable into FORTRAN and vice versa.

Non-U.S.A. computers.—Many foreign developments are worthy of mention, particularly the work of Dr. Rutishauser in Switzerland, who arrived independently at an algebraic language similar to that of FORTRAN for the ZUSE 4 computer in 1951. R. A. Brooker has developed a fully descriptive and vertically algebraic Autocoding System for the Manchester machine. Dr. W. L. van der Poel of Holland has developed some excellent methods for the ZEBRA computer, his own design, which take advantage of the micro-programming features to build up complicated algebraic routines.

CODING SYSTEMS IN PROCESS

The picture looks very good indeed for the systems in process. The majority of these are entirely algebraic and, if not directly usable on other machines, are at least mappable into each other by intermediate translators. Thus we will find that, by various practical devices, automatic coding systems may be used across manufacturers' lines, adding greatly to the negotiability of programs. Since there are not many such systems, I will describe them in some detail. Table 2 is a summary of these in condensed form. Note that we are now forced to categorize by systems and languages rather than by machine types.

FORTRAN.—FORTRAN processors are being programmed for the IBM Tape 650, the IBM 705-1, 2, the IBM 705-3 and the IBM 709. A further extension of FORTRAN is being considered for the STRETCH computer, to be delivered to the Los Alamos Scientific Laboratories. These machines, except the 705-1, 2 and the Tape 650, are all in the "forthcoming" category, and it is pleasant to note that their automatic coding systems are to be delivered along with the first production models. Specifications will be available at least four months before machine delivery date to allow coding to be done prior to delivery, thus minimizing any losses in the transitional period. Although this imposes additional burdens upon the manufacturer's programming staff, it is nevertheless accounted a worthwhile gain.

An interesting feature of the 705 FORTRAN stems from the fact that IBM is committed, as a part of its compatible-language policy, to producing FORTRAN for the 705-3 while leaving PRINT 1 unmodified.

Owing to the active interest of the GUIDE organization in conversion problems, it was agreed to pool the IBM effort with that of at least one programmer from interested companies (A. O. Smith, Esso, General Electric Pittsfield, the Texas Co., Westinghouse Sharon, and Eastman Kodak) and to consolidate the planning for the two systems so that both processors might be written with an expenditure roughly equivalent to that required for 1.3 independent processors. Initial specifications will be modest, since there is no need for optimum assignment of index registers, as in 704 systems, or for Monte Carlo optimization of object programs. AUTOCODER is well suited to be output for this processor because of its macro-instruction capability and its open-endedness. A later version will allow the output macros to specify autopoint arithmetic, determined by the record definitions of the data themselves: here FORTRAN ceases to be a purely scientific language and becomes useful for commercial problems as well. With the addition of more generators and additional superstructure in the language, it also ceases to be FORTRAN and becomes COMTRAN as we have envisioned it.

The FORTRAN language will be modified to a new level. That for the 705 is designated F5; F9 is for the 709. These two languages will be basically identical. In addition, old F4 programs and FOR TRANSIT may be run on these machines through the medium of pre-processors which convert to the revised language. Such a pre-processor may be used as an entity or incorporated in the more sophisticated processor. One of the new features in the language will be the ability to name and define sets of instructions. Thus a programmer may write a main line of coding which duplicates the *logic* of a flow chart block for block, decision for decision, while the actual subprocesses which represent the contents of those blocks are coded separately. With the F9 processor one may bypass index-register optimization at will, since this process can consume up to 80 per cent of compiling time. Registers will be assigned in rotation for quick processing and a trial run with real data. It is wasteful to expend such time if there are errors in the source program or if the mathematical techniques used are unsatisfactory. If the resultant object program is correct and suitable, one may effectively turn a switch to "Optimize" and reprocess for the most efficient object program.

Output of the 709 FORTRAN will be into the official SHARE assembly program called SCAT (Share Compiler, Assembler, and Translator). This system is being coded by IBM's Applied Programming Department from SHARE Committee specifications and is due for completion by March, 1958. With one exception it is pretty much an updating of the SAP assembly for the 704. The exception is the "Load-and-Go" technique, whereby corrections are always made by fast symbolic reassembly with the previously assembled output, which is maintained in a condensed symbolic binary form. Current good practice is to save the expanded data generated by an assembly or compilation even when corrections are to be made. SCAT does not as yet enjoy the "Define Record" characteristics, literal handling facility, or complete freedom of subroutine levels that many other existing systems have.

IT.—The IT language is also showing up in future plans for many different computers. Case Institute, having just completed an intermediate symbolic assembly to accept IT output, is starting to write an IT processor for UNIVAC. This is expected to be working by late summer of 1958. One of the original programmers at Carnegie Tech spent the last summer at Ramo-Wooldridge to write IT for the 1103A. This project is complete except for input-output and may be expected to be operational by December, 1957. IT is also being done for the IBM 705-1, 2 by Standard Oil of Ohio, with no expected completion date known yet. It is interesting to note that Sohio is also participating in the 705 FORTRAN effort and will undoubtedly serve as the basic source of FORTRAN-to-IT-to-FORTRAN translational information. A graduate student at the University of Michigan is producing SAP output for IT (rather than SOAP) so that IT will run on the 704: this, however, is only for experience; it would be much more profitable to write a pre-processor from IT to FORTRAN (the reverse of FOR TRANSIT) and utilize the power of FORTRAN for free.

UNICODE.—Remington-Rand St. Paul is writing an algebraic compiler called UNICODE for the 1103A. This is apparently a large-scale effort like FORTRAN and may be expected to operate by October, 1958. The language, too, is FORTRAN-like, and AT3 may be considered a subset of it. Many of the characteristics of the F5–F9 language exist in UNICODE. This will provide the big algebraic system presently lacking for the 1103A.

AFAC.—Allison G.M. is writing its 704 compiler for the 705 with the prime intent of obtaining compatibility. It is a commentary upon the unfortunate lack of communication in this field that Allison justified the original writing of AFAC for the 704 by stating that, although they knew IBM was producing FORTRAN, they nevertheless needed a common language for the 705 as well. Had a cooperative effort for 705 FORTRAN been started sooner, they might have spent their large effort in such a way that all would be benefited.

KOMPILER 3.—This program, for the 704, is being written to serve the special needs of the University of California Radiation Laboratory at Livermore. It is FORTRAN-like, but it implies a sharp criticism of the lack of sufficient mathematical characters in today's computers by coding each algebraic statement in three lines (or punched cards). Thus the superscripts and subscripts stand out from the main statement. This eliminates a great deal of the otherwise necessary parentheses and special notation, although the total effect is an *increase* in card volume for a given program.

MATRIX MATH COMPILER.—This program is an adaptation, by the Franklin Institute, of several previously separate UNIVAC service routines into one extensive package. Two installations are using the system with a preliminary manual, and a final manual and system are expected by January, 1958.

GP.—A compiler of the GP (Generalized Programming) type is in process for the LARC computer. This is a very large effort; certainly more than fifteen people are on the project. Algebraic coding is allowed as an instruction form, and generalized subroutines may be selectively generated for minimization in specific cases. Another interesting aspect of GP is the DuPont effort in rewriting the basic UNIVAC compilers in GP for more generality and easier expansion.

PHILOSOPHY AND TRENDS

The preceding should have indicated (allowing for a slightly scientific bias) what the existing trend is in automatic coding. Evident characteristics are these:

1. The language of communication will be our own—mathematical notation as far as possible and then English when we run out of concise symbolism. Present logical language is weak, and I imagine that even the commercial people had better brush up on their Boolean algebra. The area of loop control and recursive operations is still not well handled in existing mathematical notation, but computers are forcing the development. As an example, note the "Replace" operator (⇐) of K. Zuse. Although the ultimate in language does not exist yet, we can console ourselves meanwhile with compatible (as against common) language. There is much current evidence that existing algebraic languages are all mappable into one another by pre-processors, although these may be of varying and perhaps prohibitive complexity. The Germans, in particular, are concerned that such mappability be guaranteed before they make heavy coding investments for the many machines they will be operating.

2. The trend is to on-line system control, with the automatic coding processor always available to the running program on call. Today, this technique involves losing (during object time) the services of one or more tape units, but random access memory is much more suitable for this purpose. Eventually a replaceable photographic plate should serve this purpose in a semi-interpretive mode. Such on-line control allows primitive learning and self-improvement of programs by the computer itself in a servo process. Actual portions of object programs would be compiled only upon demand, on an exception basis. IBM and Carnegie Tech are both formulating such compilers with executive control.

Extensive means will be available for multiple-processing of intermingled test runs, compiling, and production. The CORBIE system of the National Bureau of Standards and the General Motors Supervisory System are advanced concepts in this direction.

3. The trend is to set-notation whether for data, instructions, or conditions. Both macro- and micro-instructions will come into wider usage, and machine language will be recognized as merely that subset of a given machine's instructions which happens to exist in the form of circuitry. Although on opposite sides of machine language when plotted against complexity, micro- and macro-instructions can be machine-independent for easy interchangeability. Flow-charting will become synonymous with the writing of the main line of program statements when processors consider sets of sets of instructions by name only. Minor variations in machine configuration will be handled through macro-instructions. Pro-

grams tailor-made for each member of the configurational family will be constructed by assembling an identical program *n* times with *n* different libraries of macros.

4. The trend is to standard machine configurations. The time is past when a tailored configuration could be sold for each different application. We simply cannot afford the manpower to make a different version of all automatic coding systems for all possible combinations. Some variants can be generated, it is true, with macro-instructions, but complete freedom is gone. IBM is specifying standard configurations for which automatic coding systems will be available in ample time for customers to weigh this consideration. In most cases the large savings in programming costs realized by using these systems far outweighs the cost of additional equipment to bring a machine up to the minimum. Of course, local modification by the customer for a lesser machine is always possible.

5. Future systems will gradually blend into a combination suited for both scientific and commercial work. When you see a 705-3 AUTOCODER user suddenly slip into FORTRAN in the middle of coding a payroll problem, you will see what I mean.

6. Internal computational methods are fairly well handled at this time. The emphasis at the moment is on getting much better coding for handling input and output. the preparation of reports, and file maintenance. I have concluded that people now engaged in scientific programming have a very complacent attitude, perhaps by virtue of being prior in the field. I was recently accused of being "futuristic" for recommending that an output-report generator be constructed for the 709 by the SHARE group. Fortunately, this had been demonstrated by the General Electric Hanford Report Generator for the 702, a couple of machines back. The next step for scientific users is to get adjusted and learn the many techniques developed by the business and data-processing people. Input editing, file maintenance. and report generation remain relatively unknown techniques to the scientific user. and, although he will decry this with specious arguments, he nevertheless needs them badly. He can learn much from existing business systems about basic assembly features, generators, diagnostic back-talk, macro-instructions, etc.

PRODUCTION OF AUTOMATIC CODING SYSTEMS

There appear to be three inescapable facts about automatic coding systems as we know them. They are:

1. They are always getting more complicated and will require more initial manpower in their production in order to save much greater manpower expenditure by users.

2. Just as a computer does, they require maintenance and improvement long after initial production.

3. They must be constructed open-endedly. without machine-oriented coding tricks. so that they may be adapted to different models of the same machine and

converted to future machines with a minimum of recoding. From this time on, all new systems should be additive.

Let me elaborate these points with examples. UNICODE is expected to require about fifteen man-years. Most modern assembly systems must take from six to ten man-years. SCAT expects to absorb twelve people for most of a year. The initial writing of the 704 FORTRAN required about twenty-five man-years. Split among many different machines. IBM's Applied Programming Department has over a hundred and twenty programmers. Sperry Rand probably has more than this, and for utility and automatic coding systems only! Add to these the number of customer programmers also engaged in writing similar systems, and you will see that the total is overwhelming.

Perhaps five to six man-years are being expended to write the Model 2 FORTRAN for the 704, trimming bugs and getting better documentation for incorporation into the even larger supervisory systems of various installations. If available, more could undoubtedly be expended to bring the original system up to the limit of what we can now conceive. Maintenance is a very sizable portion of the entire effort going into a system.

Certainly, all of us have a few skeletons in the closet when it comes to adapting old systems to new machines. Hardly anything more than the flow charts is reusable in writing 709 FORTRAN; changes in the characteristics of instructions, and tricky coding, have done for the rest. This is true of every effort I am familiar with, not just IBM's.

What am I leading up to? Simply that the day of diverse development of automatic coding systems is either out or, if not, should be. The list of systems collected here illustrates a vast amount of duplication and incomplete conception. A computer manufacturer should produce both the product and the means to use the product, but this should be done with the full co-operation of responsible users. There is a gratifying trend toward such unification in such organizations as SHARE, USE, GUIDE, DUO, etc. The PACT group was a shining example in its day. Many other coding systems, such as FLAIR, PRINT, FORTRAN, and USE, have been done as the result of partial co-operation. FORTRAN for the 705 seems to me to be an ideally balanced project, the burden being carried equally by IBM and its customers.

Finally, let me make a recommendation to all computer installations. There seems to be a reasonably sharp distinction between people who program and use computers as a tool and those who are *programmers* and live to make things easy for the other people. If you have the latter at your installation, do not waste them on production and do not waste them on a private effort in automatic coding in a day when that type of project is so complex. Offer them in a co-operative venture with your manufacturer (they still remain your employees) and give him the benefit of the practical experience in your problems. You will get your investment back many times over in ease of programming and the guarantee that your problems have been considered.

PANEL DISCUSSION

Engineering and Research Applications

October 25, 1957

Moderator: P. C. HAMMER, *University of Wisconsin*

Participants: R. P. RICH, *Johns Hopkins University*
E. B. GASSER, *The Toni Company*
E. L. HARDER, *Westinghouse Electric Corporation*
A. L. SAMUEL, *International Business Machines Corporation*
L. U. ALBERS, *National Advisory Committee for Aeronautics*
E. H. CLAMONS, *General Mills, Incorporated*
R. W. BEMER, *International Business Machines Corporation*
P. KIRCHER, *University of California (Los Angeles)*

P. C. HAMMER: Much has been discussed, and very profitably—the proving of theorems, for example, and the various applications in which I know you are all interested. One thing which has been left out in this discussion is the question of the output and its effect on human beings. For the Stretch machine, which was designed for the Los Alamos Scientific Laboratory, a year ago I proposed that the output should be in the form of moving pictures of surfaces. The fact is that the human mind is incapable of grasping numbers in large quantities. We could swamp all the faculty members at Wisconsin with our puny little 650. They could not read what we could put out even if they were so inclined. Numbers are singularly poorly adapted to the human mind. Curves are a little better for interpretation. If a person has to act on information, he has to read it; the way it is now, the opportunity for reading information is far less than the power of machines to put it out.

Another point which bears on the use of computing but is not really an application of computing is the question of mathematical research in connection with the methods we use. We are using horse-and-buggy mathematical methods in a machine age. For example, there is a feeling among many people, largely due to the existence of methods in large quantities, that finite differences will be the answer to differential equations in the future. This probably will not be so. There is no hope that I see now for really doing a good job on a partial differential equation which is the honest flow problem: four independent variables (that is, three spatial and one temporal) characterizing a fluid flowing in space. To do this by finite differences would be almost incredible. It would be far more incredible if, after obtaining a solution by that method, a function table of four independent variables were to be printed as output. You could not read it or understand it if you had it. The entire output situation, I would say, is rather unsatisfactory.

It is also unsatisfactory to consider more and more automatic programming techniques when we do not know what we are going to do with these things in the future. For example, the kind of thing which Dr. Samuel mentioned, along the lines of proving theorems, is perhaps going to take hold one of these days, and maybe algebra will be done by machines. It is not known whether or not the automatic compiler which is devoted primarily to arithmetic on the assumption that all you handle is digital numbers in the machine will be suitable for this. We are not reflecting enough on the possible use of these computers: what they might be used for if we could ignore the cost momentarily. It is important to ignore the cost in order to get an adequate theory.

Now, to turn this discussion over to some of the other people, I want to ask one question of Dr. Rich. He mentioned that two thousand words of storage was about the minimum and maximum size for a program. I was wondering if he meant that there was a human incapacity to do more than this or a machine incapacity to handle more than this? I would say that there are physical systems which cannot be done with that number of steps.

R. P. Rich: I would like to start emphasizing a number of boundary conditions I put on this theorem. One needs hypotheses to draw conclusions. In the first place, the total storage was not two thousand words, but rather seven thousand, where five thousand words were used for storing tables, constants, and other reusable information. Two thousand words is the actual running program, consisting of the instructions executed each time around. That was the first restriction.

The second restriction was that this was for a particular kind of problem; this working storage figure would obviously be very different for other types of problems. That is, should one try to do certain nuclear-reactor problems within a factor of ten of this amount, difficulties would arise. The real point that I meant to emphasize by overstating my case (as I think one must to get points across in so short a talk) is that it is not only possible but also easy and obvious to ask the machine to do a lot of things that you should not ask the machine to do. For example, if you know the effect on the output of certain of the inputs, then these should not be redetermined during every run by Monte Carlo. In other words, if analytical answers are available for parts of such problems, then these should be inserted analytically. Random numbers should not be drawn to see whether a sine wave is in fact a sine wave. That was the major point I wanted to make—that random Monte Carlo procedures should be restricted to the things one actually does not know how to do analytically.

Another point I tried to make was that, if sampling procedures are used, then, in order for answers to be precise enough to be of any use, fairly large samples must be run. With machine time costing what it does, and the demand on the machines being what it is, in order to get a reasonably large number of samples, the time per sample on the machine must be reasonable. Therefore, if too much is required of the machine at each single calculation, then each answer is a good answer, but it still is only one point of a sample; a hundred-point sample with several approximations at each point may still be worth a factor of three better than a ten-point sample, just because of the sampling error. I could go on with this subject at length, but I am sure that that is not the best way to spend our short remaining time, so I will let it pass for now as a rough answer to Dr. Hammer's question.

P. C. HAMMER: Dr. Harder, you mentioned in the course of your talk that there were certain instances in which network calculators and digital computers were equally efficient. I wonder if you could give us some of the qualitative characteristics of such a situation as that?

E. L. HARDER: For a system-stability calculation it is necessary actually to invert the complex impedance matrix of the power system. In this case the inversion of the matrix requires considerable time and must be carried out for each network configuration involved in the stability study. This is a type of power-system problem in which I think the network calculator excels at the present time.

At present, for load-flow problems such as I illustrated, the efficiency of the two machines is about the same. Practically all the load-flow studies are being done by the nodal equation approach, and this has several effects. For one thing, considerable network identity is lost, and the mutual impedances between lines cannot be taken into account.

In short-circuit or ground studies, a third type of power-system problem, there are mutual impedances between lines, and so, to use a digital computer technique, it is necessary to start all over again, not with the nodal equations (that is, Kirchoff's current law), but rather with Kirchoff's voltage law. It is much harder, however, to set up the voltage equations in a systematic fashion. Whether the digital computer or the network analyzer is better often depends on such a small thing as how good the initial guesses are. In a case where some experimenter has found that he can run the problem let us say two to one cheaper by digital computer, on digging into it one might find that, had his initial guesses been poor, it might have been two to one more expensive. So the question Dr. Hammer asked can be answered only by using particular instances.

P. C. HAMMER: Dr. Samuel gave a very interesting talk about the proving of theorems. I wonder if he could give us an idea of what he considers the most promising line of thinking right now? I have been thinking about proving theorems in a very modest sense, not in the ambitious sense Dr. Samuel has been describing. For instance, I have run into some simultaneous non-linear equations, twenty-three hundred of them, and I would like to solve them. (Twenty-three hundred is one step on the way up to infinity.) The simple kind of thing I am thinking of is the proof by induction. Mathematical induction is one case in which one quite often knows fairly well the kind of thing he is doing, and therefore he could in principle carry out the proof by complete induction, by using roughly the same type of thing Dr. Samuel suggests. Let the machine establish the guesses as a function of N, in a one-index problem; and let the machine prove that the guesses work. This could be done in cases where the only excuse for doing it by machine really is the manipulation, the impossibility of writing out as many equations as one has. You cannot see them. So you know what to do, but you cannot do it, not practically. The same thing applies to calculating. You could presumably do all the calculation that the machine does, but you cannot quite carry this out. Would this kind of theorem, or theorems in complex functions or topology, be the kind of thing you might be able to prove?

A. L. SAMUEL: It is very easy to talk glibly about these things, but at the present time plane geometry is proving extremely difficult, to say the least. I do not know when we will get to more complicated things such as those to which you refer.

I might make one general remark, which may be obvious to you, in respect to dealing with problems both syntactically and semantically. If you are proving a theorem, you use the semantic interpretation of the numerical data as a guess to tell you whether you are proving your theorem or not. There is a converse of that, in which you are trying to prove a theory—trying to prove that a present theory is not true—where exactly the opposite should be done. You should use the semantic information as the truth, trust it, and doubt the syntactic information. I think we may be actually using computers before too long to derive new theories, which is just the opposite of proving theorems.

P. C. HAMMER: Does anyone in the audience have any questions to ask the speakers?

H. H. KANTNER (*Armour Research Foundation*): I would like to ask, "What is the relationship between automatic programming and symbol identification or, as we know it more customarily, character recognition?"

R. W. BEMER: I hope that eventually we will have a typewriter for input, similar to a typesetting machine or linotype, and such that when you write out an equation it can be recognized by the computer. Presently we are limited in the scanning of an algebraic statement, or even an English statement, by the many extra symbols used to separate our meaning. If I had the right kind of typewriter—imagine it has plenty of characters, upper- and lower-case roman, upper- and lower-case Greek, big and little numerals, brackets of different types, even hands that point, in short, a multiple font—I could press a button which would put the platen at half-carriage and cause a bit in a control word which says, "This is subscript." So I could subscript by sin X, or by reverse I could superscript, and I could go up to many levels of superscript. In this sense I think we will eventually be able to take type-set information and feed that into the computers. I do not know about handwritten information. That seems pretty far away.

H. H. KANTNER: But you take the elements of the equation as symbols by themselves, and yet the equation per se is a symbol, as pictorial display.

R. W. BEMER: Maybe this will be pertinent. In the future system we will be able to give temporary connotations of meanings to any variables, or symbols, or sets of symbols. Thus. for temporary purposes, I can say that these operations will all be double-precision arithmetic. and these will all be complex arithmetic, from here on. Further. one can say that plus and minus indicate matrix operations instead of plain linear operations; or. if I wish to replace sets of items, I can say, for example. that all variables which have names starting with "P" are in South America. One can define this to the machine and change it at will. It is simply a matter of altering the table look-up in a dictionary by classes or sets.

H. H. KANTNER: I have not got an answer to the question, "What is the relationship between symbol identification and automatic programming?"

P. C. HAMMER: May I take a stab at this? In a generalized sense, I think you could say that symbol identification is the same type of thing as automatic programming. That is to say, your machine gets a certain word, it recognizes a certain word, like sin x. It proceeds to operate on this and generate a sequence of responses to it. If you call the whole sequence of responses a "transformation" and consider the whole code you are putting in as a symbol, then, if you want to, you can say that this is one symbol and the machine responds to the whole symbol—you can say that this is the same as character identification. The machine knows what to do with this whole thing and recognizes the whole code altogether. I am thinking of it as one symbol now. What is a symbol anyway? It could be the whole thing. So I think the answer is that automatic programming is the same as symbol identification, presuming the machine makes no mistakes. Does that answer your question?

H. H. KANTNER: Thank you. It is a very good stab.

R. S. DIKE (*Caterpillar Tractor Company*): Along the same line, I would say that, if you think about this reflectively. you will realize that the Chinese had a very strong symbol language, which has become very complex. Would we not be better off sticking with the simple symbols of our own language and stringing them out, rather than making too complex a symbolism? Would we not be forcing ourselves to have a language too complex to handle?

L. U. ALBERS: I believe that you certainly can go too far in taking advantage of all the logical symbols and subscripts and German characters and so forth. In the direction of Dr. Hammer's suggestion, if you can present people with curves or pictures or written decisions or judgments, this is probably much more helpful than just multiplying the language.

Incidentally, I would like to mention one other thing in regard to the matter of designing machines to learn. A young student at Case has programmed the 650 to learn to play ticktacktoe. It starts off not knowing how, and in the process of seventeen or eighteen tries it has learned and is capable of tying or beating anyone it plays.

A. L. SAMUEL: I am sorry I did not mention that there have been many, many attempts of this sort. A man at the National Physical Laboratory in England programmed the Ace to do the same thing. There have been a lot of things done;

I picked a few isolated examples by way of illustration. I apologize to all the people I did not have time to mention.

P. C. HAMMER: In a way, the computing industry started with the big machines. The government was behind these things, and the big companies started making big machines. Now we have this business of interpolating to zero with smaller types of machines. I was wondering if Mr. Gasser might say something about whether or not his company is going to contemplate getting bigger ones or is going to hold fast to this order?

E. B. GASSER: In my letter I sent forward to Boston, recommending the purchase of the LGP-30, we stated that at the current level of research activity we would be satisfied with this machine for the next four years. At the end of this time, I think we will take a good, hard look at our machine and the competitive machines that exist at that time and make further decisions. We are bound for only four years. I think that the way things are going in the Toni organization, with the strengthening by Gillette of the central research organization, we will find ourselves in the market for a larger computer by that time.

E. L. HARDER: I would like to ask Mr. Bemer a question in connection with the translation from machine to machine. Of course the translation between the superlanguages, and then the compiling for particular machines, is a fine way, provided you program in the first place in one of those superlanguages. But can you comment on what is going to be our ability to use the programs for the present machines on their successors?

R. W. BEMER: The only way this is possible is on a machine that is specifically designed to accept all instructions of a previous machine plus additional instructions: such a computer is the 705, Model 3, which will handle all programs written for the 705, Models 1 and 2.

As a general procedure I do not think that it is possible to swim upstream to the general language. If you take a specific machine and take some odd-ball characteristics it has—divide by an alphabetic number and swap the result end for end, for example—anyone can produce a program that nobody else can ever figure out in terms of what it was intended to do. If you want to compute J_0 (x), I can write down a program for a particular computer: but I cannot look at a program for SWAC, say, and know that it will give me J_0 (x).

E. L. HARDER: What about 704 programs on the 709?

R. W. BEMER: There is a special device that will enable you to run all 704 programs on the 709. I think that it is only necessary for the conversion period, for the machine-language programs.

E. L. HARDER: What can be done if you go to a large core memory and want to drop the magnetic drum memory? If you have a 30,000-word core storage, it seems foolish to retain an 8,000-word intermediate speed storage unless you really need it for some special reason. Do you think that the existing programs for drum and a 4,000-word, high-speed memory can be translated mechanically in some way, perhaps by using tabulating equipment, to a machine that does not have a drum?

R. W. Bemer: I think that it depends upon the complexity of the program. For instance, Dack Ridgeway could tell you that he could not find any way of mechanically converting Fortran, written for the 4,000-word, high-speed memory and the 8,000-word drum, so that it would work with the 32,000-word, high-speed storage. They had to get in and change table sizes by hand; they had to juggle it. I think that there are many, many things like this that we are going to be stuck with for a period of time, which we cannot afford to mechanize, because writing the program for mechanization would be more work than it would be to do the original problem. We are faced with many difficult problems in going up in the hierarchy of machines; the only solution I can see is getting to a high-level synthetic language quick enough so that we do not engender much of this machine-language instruction. We could then hope to get the problem out of the way once and for all.

R. P. Rich: You mentioned a five-to-one reduction in time in going to Fortran, but you did not say from what. Usually this factor is ten to one instead of five to one, but a person never says from what.

R. W. Bemer: I am trying to be conservative. It does not become me usually. This is from the SAP language, which is symbolic machine language. If you were to program in pure binary, you would find it twice as difficult as SAP, so we multiply the two together and say that Fortran reduces ten to one in programming over actual machine language.

R. W. Floyd (*Armour Research Foundation*): Most equipment seems to be designed around pure numerical or alphabetical input, whereas most mathematicians are trained to use Greek letters for angles and special signs for "greater than," "equal to," "minus," and so forth. Is anything being planned for coming machines or interpretive routines to make it easier for the poor mathematician?

R. W. Bemer: The Los Alamos people have their own design for a 300-character typewriter which they would like as input to the Stretch system. They made a request for price quotation to IBM to reproduce this, and it is under consideration now. It has pretty much everything on it. I am very glad to see this myself.

A. Opler (*Dow Chemical Company*): Yesterday, Dr. Hopper indicated that Remington-Rand, to solve this input problem, wanted to go to the English language instead of to symbolic. I think your approach is opposed to hers. She seems to have a broader brush treatment in the sense that she could meet business needs as well as those of the mathematician. The mathematician still knows the king's English.

R. W. Bemer: I do not know how the king got into this. It is the queen's English now. Anyway, I do not think that actual language as such will hold up too well if you extend Dr. Hopper's principle indefinitely to all the different ways one can express statements. She says that, after perhaps twenty-nine different combinations, you have reached the limit of what anybody could possibly say. In the more complex programs there will undoubtedly be found other ways to say the same thing to the machine. I would much prefer a mathematical or symbolic logical notation, which consists of specific characters, and see this used as an unambiguous input to the machine.

A. Opler: There are 20,000 characters in the Chinese language.

R. W. Bemer: I do not propose to go to the Chinese language. We tried to make up a list once, and within the neighborhood of 180 different characters we could do just about anything anyone required for the majority of scientific or commercial work. It seems reasonable that two four-bit characters in combination can express all the alphabetic, numerical, and special-character symbols we will need for a while.

A. L. Samuel: One of the problems with English is that a person can make a statement in English without realizing that he is not being precise. This is one of the characteristics of the English language. So there is a certain argument in favor of requiring a limited artificiality in machine language which forces the person stating the problem to recognize what he does not know about the problem in order to state it precisely.

Question from the Floor: There are a number of installations I know of that have large-scale computing equipment and also have some of the smaller machines that have been described today. I would like to hear from some of the panel members what they anticipate the effects of automatic coding techniques will be in terms of what uses these smaller machines will have in those installations with the larger equipment available?

E. L. Harder: We have this problem at Westinghouse in that we have forty plants and five large-scale computers and also about a half-dozen medium-scale computers. There is a possibility of the small computer's being used right along with the large computer. There is a possibility in a big plant of its being used two or three floors away or two or three buildings away; and, of course, there is a possibility of its being used in a plant that does not have any other computer. Now, where there is a large-scale or medium-scale computer present, then the decision as to whether you need a small computer also, for smaller problems, is in a way competitive with the use of automatic programming. We are conducting an experiment right now to try to test this out—a very simplified version of Fortran as compared with the use of a small computer. The two factors that enter in are convenience and the time to do the programming. Of course, the cost comes in, too. In general, automatic programming seems to be about twice as fast as the programming for most small computers, which do not have the benefit of symbolic programming. So the programming is a little shorter with the automatic programming technique. We are still trying to work out the balance of this gain for the automatic programming as against the scheduling difficulties and the problem of getting the answers for small problems through big computers, scheduling them along with other work. We do not know the answer yet. We rather feel that there will be a lot of use for the small computers. In fact, in our company now we have five of them, and a number of others ordered, along with the big ones. Most of them are in special-purpose applications, although in one plant there is a small computer in the same office with a medium-scale computer. We are trying to work out the optimum pattern in order to advise our various plants.

R. W. Bemer: I would like to toss in a little remark. I am in favor of the short-order-cook policy that I think will come into effect perhaps five or ten years from now. It might resolve at least a certain class of problems as between the small and large computers. If one had an extremely large, extremely fast centralized computer with various lines radiating out, and with terminal facilities such as a person now only gets in the form of input-output devices at the computer. and if one could have high-speed transmission to and from this centralized computer, it would be like a short-order cook. It takes the orders off the lines and. so to speak, heats up the griddle and sees that the toast is ready while it is pouring the coffee. It will be self-scheduling, self-regulating, and self-billing to the customer on the basis of use of the input-output device. I think, since the larger and faster computers, as far as production problems are concerned, always produce more problems solved per dollar once the problems are in the machine, that this is the obvious direction to go. I agree that at the present time there are many small computers that seem to take less trouble than a large one; but I think that, in the long run, we will use the largest computers and will start thinking in terms of compatibility of languages and ultimately in terms of a single language.

P. C. Hammer: It is getting a little late. Maybe we had better draw this to a close. If there are no more questions. I want to thank all the speakers for the excellent job they did today, and thank you all for being here.

10

I *NEWS*

The GUIDE organization of 705 users is sponsoring a cooperative project with IBM to produce a FORTRAN system for the 705 models I and II. The processor will be written with the conversion to 705 model III in mind so that the total effort will be minimized. Furnishing programming people for this project are:

Westinghouse Electric
Eastman Kodak
General Electric Co.
Standard Oil Co. (Ohio)
The Texas Co.
A. O. Smith Corp.

The first working version is expected to be tested by August 1958. The FORTRAN language which it accepts will be identical with that for the 709.

II *CONTRIBUTIONS*

To start, and by way of demonstrating how trivial contributions may be, I am showing one taken from the PRINT I system for the IBM 705, a serial, decimal, VET machine.

A MACHINE METHOD FOR SQUARE-ROOT COMPUTATION

R. W. Bemer
I.B.M. Corporation, New York City

Computers with operations having variable execution times (VET) require a different class of subroutines to take full advantage of these characteristics. Well-suited for computing square root on decimal machines is a variation of Newton's method which uses a linear first approximation such that convergence to the desired accuracy occurs in 2 iterations, thus causing a fixed and predetermined execution time.

Floating point square root routines operate on arguments of the form:

$$N = M \cdot 10^{P} \quad \text{where} \quad .1 \leq M < 1, \text{ and M is always positive}$$

To establish a common program for both odd and even powers (P) of 10, let

$$N = m \cdot 10^{p} \quad \text{where} \quad .01 \leq m < 1, \text{ and p is always even.}$$

Then, $\sqrt{N} = \sqrt{m} \cdot 10^{(.5p)}$ and $.5p = .5P\ (+\ .5 \text{ when P is odd})$

$$m = \begin{Bmatrix} M \text{ (if P is even)} \\ .1M \text{ (if P is odd)} \end{Bmatrix}$$

Iterating twice with initial approximation A_1,

$$\frac{m}{A_1} = Q_1 \qquad A_2 = .5\,(A_1 + Q_1) \qquad \frac{m}{A_2} = Q_2 \qquad \sqrt{m} \cong .5\,(A_2 + Q_2)$$

But $\quad m = A_2Q_2 = A_2(A_2 + \Delta) = A_2^2 + A_2\Delta$

$$\frac{m - A_2^2}{2\,A_2} = \frac{A_2^2 + A_2\Delta - A_2^2}{2\,A_2} = \frac{\Delta}{2}$$

$$\sqrt{m} \cong \frac{A_2 + Q_2}{2} = \frac{2\,A_2 + \Delta}{2}$$

$$\sqrt{m} \cong A_2 + \frac{\Delta}{2} \cong A_2 + \frac{m - A_2^2}{2\,A_2}$$

This form is designed to minimize the number of digits of quotient which must be developed. Users of desk calculators will recognize it as the standard method of developing half of the required accuracy by long-hand square root method and dividing to place the second half of the root in juxtaposition. This method need not be followed, but the tables of segments of approximation still apply..

The starting approximation A_1 is derived by using a table of linear segments which approximate $\sqrt{m}$ within a prescribed tolerance. This tolerance T is a function of m and the allowable error ϵ in the final approximation. To compute T:

$$\frac{m}{A_1} = Q_1 = \frac{m}{\sqrt{m} + T_1} = \sqrt{m} - T_1 + 2\epsilon$$

$$m = m - T_1\sqrt{m} + \sqrt{m} - T_1^2 + 2\epsilon\sqrt{m} + \boxed{2\epsilon T_1} \longleftarrow \text{Ignore}$$

Therefore, $2\epsilon\sqrt{m} \cong T_1^2$ and $T_1 \cong \sqrt{2\epsilon}\sqrt[4]{m}$

By extension, for a second iteration, $T_2 \cong \sqrt{2T_1}\sqrt[4]{m}$ Total tolerance $T_t = T_1 + T_2$

For 8-digit accuracy, $\epsilon = .5 \times 10^{-8}$ and $T_1 = 10^{-4}\sqrt[4]{m}$

For 10-digit accuracy, $\epsilon = .5 \times 10^{-10}$ and $T_1 = 10^{-5}\sqrt[4]{m}$

Working within these tolerances, tables of linear segments may be constructed. The number of segments is minimized for a given accuracy, simultaneously with minimizing the number of digits in the multiplier and constant term for least execution time. Tables for some common accuracies are shown here.

$A_1 \cong aX + b$ where X = most significant part of m

8-digit			10-digit			12-digit		
Range	a	b	Range	a	b	Range	a	b
.01–.02	4.1	.060	.01–.02	4.2	.0585	.010–.014	4.58	.05439
.02–.03	3.2	.078	.02–.03	3.1	.0803	.014–.020	3.84	.06482
.03–.08	2.2	.110	.03–.05	2.5	.0991	.020–.028	3.23	.07712
.08–.18	1.4	.174	.05–.08	2.0	.1240	.028–.040	2.72	.09153
.18–.30	1.0	.247	.08–.13	1.6	.1545	.040–.056	2.29	.10876
.30–.60	0.8	.304	.13–.23	1.2	.2060	.056–.076	1.95	.12781
.60–1.0	0.6	.409	.23–.39	0.9	.2749	.076–.105	1.66	.15004
			.39–.60	0.7	.3550	.105–.145	1.42	.17548
			.60–.84	0.6	.4148	.145–.195	1.21	.20596
			.84–1.0	0.5	.5005	.195–.260	1.05	.23745
						.260–.350	0.91	.27395
						.350–.470	0.78	.31953
						.470–.630	0.68	.36649
						.630–.820	0.59	.42301
						.820–1.00	0.52	.48005

Note that range limits are also chosen for minimum number of digits, for minimum TLU time.

Slight adjustments to these tables are possible. Because of the unequal interval in arguments, address modification from the argument is usually impractical; the normal method is to truncate the argument to .xx or .xxx unrounded and do a table lookup on comparison. Because of rounding overflow the subroutine is easier to write if the $\sqrt{.99999999}$ with even power maintains the same mantissa for the square root. An early test for this condition will usually save program steps inasmuch as overflow is guaranteed never to occur. Other tables could be constructed for different accuracies, but if more than 12 digits are needed it will probably be better to use 3 iterations. A similar method is also possible for binary machines with VET.

Example: Find the $\sqrt{.12345678}$

$A_1 = 1.4\,(.123) + .174 = .346 \ldots\ldots\ldots$

$A_2 = .5\,(.346 + .35681) = .35141$

$.12345678 \div .346 = .35681$

$.12345678 \div .35141 = .351318346$

$\sqrt{\ } = .5\,(.35141 + .351318346) = .35136417$

or,

$$\frac{.12345678 - (.35141)^2}{.70282} = \frac{-.0000322081}{.70282}$$

$- .00004583 + .35141 = \sqrt{\ } = .35136417$

11

XTRAN ANNOUNCEMENT - SHARE Meeting - 26 Feb 1958

XTRAN is a tentative name for a tentative source language which is to superstructure on the existing FORTRAN language. Certain of the preliminary specifications are outlined on the sheets you now have, together with some coding examples to demonstrate certain salient features. Note that these specifications are incomplete, particularly with respect to input-output and logical statements. This does not imply that we do not have improvements developed, but merely that we could not decide on a proper form for this presentation, rushed as we are.

XTRAN follows a method demonstrated to be feasible by the FOR TRANSIT system, which is a means of running source programs in the FORTRAN language on the 650. This is done by means of a source language - to - source language processor from FORTRAN to the IT language, which is then used to produce symbolic and eventually machine language coding. Thus XTRAN programs produce FORTRAN programs which are then fed into the FORTRAN processor for the 709. Again, the standard SHARE machine is the configuration used thruout.

The basic mechanism of XTRAN is the Pre-Processor program, which is being currently flowcharted for the 709. This pre-processor is itself a multi-pass intelligence-gathering and transforming unit quite similar in principle to the multiple passes of the PACT system. I believe that most of our future programming systems will be constructed on this modular principle, limiting the variety of functional work performed on each pass so there is little conflict, and certainly more flexibility for improvement and change. Thus the information gathered in passes 3 and 5 might show that there were no source entries requiring the services of pass 8, which would be eliminated accordingly. For processing XTRAN on the 705 models I, II and III, no pre-processor as such is required because the combined GUIDE-IBM working group is starting from this language and translating directly to Autocoder III.

During the pre-processing, many of the XTRAN statements will produce multiple statements of the FORTRAN variety, so that the program is likely to be much expanded when entering the FORTRAN processer. We are coordinating XTRAN work with that of the FORTRAN 709 processor so that the analysis and information-gathering done in the XTRAN pre-processor is switched off in the FORTRAN processor and not duplicated any more than necessary. If there had been sufficient time to make an integrated system before delivery of the first 709s we would have done so, but the present mode of fabrication has a greater safety factor for completion on time. In actuality we expect the overlap to be negligible.

The basic intentions for XTRAN are:

1. To minimize the amount of actual writing and coding, resulting in fewer entries,

2. To minimize possible coding errors by allowing more freedom in rules and automatically inserting new and corrective statements. Thus:

   a. Algebraic statements may have mixed expressions containing fixed or floating point variables or constants. Multiple sinks may be specified, as "RUNNINGSUM = INITIAL = some expression". This same statement may be used to create multiway programmed switches.

b. Declarative statements (as opposed to imperative statements which have to do with program flow) may be written anywhere in the program. The pre-processor automatically moves these to the beginning in the output listing and groups them appropriately.

c. Conditional transfer statements need show referents for only those statements which are not the next following. Thus IF (a) ,, THERE would mean go to the next statement if the value of $a \leq 0$. This also occurs for defining sets of instructions from the first statement to the last, in that the first need not be mentioned if immediately following. This minimizes the need to think up different names for statements.

d. Statement referents and variables may be assigned virtually ANY 10 character alphanumeric name (numbers are still a proper subset of referents). The pre-processor converts all of these to numeric equivalents for FORTRAN and assigns and creates new names as necessary. This eliminates several opportunities for FORTRAN errors, in that there need be no special names for classes of variables such as fixed point, is less danger of running over the 6-character limitation and names may be more meaningful and differentiated with less chance for duplicates.

e. Subscripting and range control on DO statements are both general and compound, allowing essentially complete freedom to use subscripts and incrementation as any fixed point algebraic expression, with the present limitation that they not contain parentheses. Here again is elimination of error source and added convenience, particularly in the use of negative subscripts and decrementing in a DO statement. Variation for all indices may be expressed in a single DO statement and each of these may have multiple stepping, as a(b)c(d)e...... (See the matrix multiplication ex.)

F. Most fixed point variables are determined to be such from their usage in the statements. Those few which may not be are so stated in a FIXED POINT LIST statement. This also applies to Boolean variables.

g. Much writing may be eliminated by the DEFINE and EXECUTE SET statements, which effectively copy groups of coding in place.

h. Many misspellings and incorrect statement punctuation will be detected and automatically corrected and ignored. Those which cannot are at least detected before they go into the lengthier FORTRAN processing, as are logical errors in coding and ambiguous statements.

3. To be compatible with lower level existing languages in that any FORTRAN statement may exist in XTRAN, but not vice versa. This is important in that there may be cases where XTRAN programming when used to the utmost would produce a less efficient and slower-running program, although programmed much more simply in source language. In this case, as with the present operation of 704 FORTRAN, the expert programmer may achieve better efficiency thru understanding of the system and coding in the lower level terms.

4. To be compatible with many IBM computers with like capability, such as the various 705s, Tape 650, etc. Machine orientation has been removed from the language and facility is provided to code specific sections in the symbolic machine language of the several computers, meanwhile maintaining a careful watch to catch and give warning if such programs are run on other computers for which these sections are unacceptable.

5. To allow the fabrication of generalized higher-level statements by the open-ended definition of new language with the appropriate generators. For this it must have recursive properties and be susceptible to set notation, symbol substitution, logical algebra, etc.

6. To allow a more nearly "flowchart" type and way of coding, where the detailed blocks of coding are filled in later although not subjected to restrictions that they must appear in the source program in any particular flow pattern.

7. To facilitate multiple processing and testing of intermingled programs

COMPUTER LANGUAGE COMPATIBILITY THROUGH MULTI-LEVEL PROCESSORS

HEMMES/BEMER

Everyone has probably heard the story of the test given to determine the aptitude of a man applying for a job as a mathematician. For the first part of the test he was placed in a room, and in this room there was a bucket of water on a table, and a stove. The problem was for him to heat the water. The solution was obvious. He moved the bucket of water from the top of the table to the stove. He then lit the stove and heated the water.

For the second part of the test, he was placed in the same room with the same props. But this time the bucket of water was on the floor. The problem was the same - to heat the water. After some thought he moved the bucket of water from the floor to the table. This was the correct solution, and he passed the test because by moving the bucket of water from the floor to the table he had reduced the problem to one he had already solved.

Now this is a silly story, and usually told in an effort to be funny, but the hero of this story was faced with a problem not unlike some problems often faced by programmers who are trying to create an automatic programming system for a digital computer. Often the programmer will already have at his disposal an existing subroutine, or set of subroutines, or even a checked out program, just as the hero of the story had the already checked out table to stove subroutine.

A programmer is faced with the choice of going as directly as possible from the source language to the machine language object program, saving as many instructions as possible and carefully optimizing, or taking advantage of the existing subroutines by writing a program that calls upon the subroutines when they are useful.

A programmer who has at his disposal more than just some subroutines has a more difficult choice. He may have access to an existing symbolic assembly, or even an algebraic compiler.

If his task is to create a more sophisticated and more powerful programming system than the existing one, he is still faced with the same choice. Just as the hero of the story could have moved his water directly to the stove, the programmer could chose the direct route of going directly to machine language, or he could take advantage of the existing system by writing a pre-processor that translates from the new source language to the language of the existing system.

There have been many arguments pro and con about the merits of the indirect route over the direct and vice versa. The direct route has the advantage of shorter processing time but is much more work for the programmer as it takes very little advantage of existing coding effort. The indirect route has the disadvantage of longer processing time, but has the advantage of being less work, since it takes full advantage of an existing system which may already have most of the features desired in the final overall system. These arguments point up one fact. THERE IS NO GENERAL RULE. The route to be taken from source language to object program depends on many things that will vary with each individual case. Some of these are:

The programming investment in the existing system and the efficiency and usefulness of the existing system.

The programmer's budget for creating the programming system and his deadline for having a checked-out running system.

These last two points are perhaps the most pertinent. If a programmer simply doesn't have the budget or the time to sit down and write a complete system from scratch, then the choice is made for him. He must use the indirect route of the preprocessor.

Another important point to be considered is that when a system is created by tacking preprocessors in front of existing automatic coding systems, what kind of monster will the final product be?

In order to help understand what this final product will look like, some points about preprocessors should be made clear. Actually what a preprocessor should be is a source language to source language translator, that translates from a higher level source language to a lower level source language, and incorporates advanced automatic programming features, during the translation.

To be fair, a preprocessor should be regarded more as the first pass of a system, or the first group of passes of an overall system, rather than as an independent programming system simply placed in front of some other programming system. True, the overall system may be initially created by this tacking on of the preprocessor, with the resulting increase of processing time and some duplication of effort. But once the system is "off the ground" and checked out, and the deadline, if any, has been met, then later rework should take over the elements of the lower system and absorb them where they belong.

The FOR TRANSIT system for the IBM 650 illustrates the successful use of preprocessors. The system came into being in this manner: SOAP, a symbolic assembly system, was written by IBM programmers and was an existing system. Dr. Perlis of Carnegie Tech. and his associates, J. W. Smith and H. R. Van Zoren, wrote a compiler which they named IT and whose output is the input language of SOAP, thus eliminating the need for them to write an assembly for the IT system, as the program was assembled on a second pass using the SOAP system. The next step was for IBM programmers to write the FORTRANSIT preprocessor that translated from FORTRAN statements to IT statements. This system is more than just a source to source language translator. The FOR TRANSIT preprocessor incorporates many automatic programming features that were not possible with the IT system alone, has a one for many input output correspondence, and in addition gives the programmer the advantage of the simpler FORTRAN language.

The resulting overall program is a three pass system that works in this way: Source programs are written using FORTRAN statements. The first pass through the 650 scans and classifies the input statements and sets up necessary tables of correspondence between FORTRAN variable names and IT variable names, at the same time double subscripted variables are assigned IT variable names. Then the whole program is translated to IT statements for output. For the second pass, the newly created IT statements are used for input. The compiler then scans and breaks down each input statement, compiles subroutines and translates on a one for many basis from the IT statements to the symbolic language of SOAP for output. For the third pass, the symbolic SOAP program is used for input, and is assembled, the output being an optimized machine language object program ready for execution.

The overall result of this experiment in tacking a processor in front of a processor that was tacked in front of a processor has been a three pass system that is long on processing time and due to the three passes, one which has been unwieldy in some cases. But creating the FOR TRANSIT system accomplished this:

The programmer with a problem to solve need only concern himself with learning the FORTRAN language and very little else. In the ideal case the programmer is unaware of whether it takes three passes or three hundred passes to process his FORTRAN statements.

While the system is being used in the field, programmers at IBM are working on a two pass system. The first pass will accept the FORTRAN statements and compile a symbolic program which is assembled on the second pass. Duplications on coding are being eliminated. The whole system will be tightened up. The result of this is that this two pass FOR TRANSIT will have a marked decrease in processing time and greatly increased flexibility as well as a more efficient object program. But remember, while this new system is being prepared, people have the original system to use, and when the improved system is distributed, there will be very little interruption, change over period, or relearning time, in the 650 installations that are using the FOR TRANSIT system.

Experience in writing and using the FOR TRANSIT system has shown that there can be many important advantages of pre-processors.

One is the possibility of introducing more advanced automatic programming features to an existing system through the use of a pre-processor. This can be accomplished by a preprocessor that not only accepts a higher level source language as input, but also contains the routines and ability to prepare this input for processing by the lower level system. The preprocessor can actually do a little programming, thus easing the load on the human programmer and making the overall system closer to an automatic programming system rather than coding system.

A second advantage is that inter-computer compatibility is possible through the use of one common source language and translators to the source languages of existing automatic programming or coding systems. The FOR TRANSIT system might be used as an illustration again here, as it translates from a 704 input language to a 650 input language, making it possible to use FORTRAN statements on either a 704 or a 650.

Another advantage is that it is possible for programming systems to exist before the final source language is specified. For example, there are few people actively using computers today who are not using some automatic coding or programming system. Yet work is still in progress by the ACM AD HOC Committee to specify a common computer language. Once this common language is specified and accepted for use, then it is possible for source to source language translating preprocessors to play an important role. This indirect approach can enable people to begin using the new ACM universal language without waiting for whole new systems to be created from scratch.

A final advantage is the obvious but often overlooked advantage of less likelihood of programming errors. Any time there is less work for a programmer to do, less for him to understand, fewer rules to follow, there will be a result of fewer errors on his part. In addition to this, the preprocessing pass should incorporate error detection as one of its features. Error detection backtalk from the preprocessing pass in the form of diagnostic print outs can be a great aid to debugging, and the earlier in the system errors are detected the more time saved.

One final point about preprocessors is that, if an automatic coding system exists, and many source programs have been written in the language of this existing compiler, and then a preprocessor is added, the existing source programs are not necessarily made obsolete. They do not have to be rewritten any more than a machine language program would have to be rewritten when someone comes along with a symbolic assembly program.

Once the overall system is completed source programs may be written in any one of the levels of the system, and the system should give the programmer the freedom of changing the mode of his source statements from one level to another within his program.

This can be made possible by building an intelligence into the preprocessor that enables it to recognize the mode of the input statements. Any input statement that is of a lower level than the highest is simply passed along to the proper place in the system for processing. The remainder of the talk was a description of the proposed XTRAN 709 system.

Automatic Control
April 1959

cc sent
Bingham

Evaluating Intelligence for Programming Systems

A remarkable variation exists in the degree of sophistication of various programming systems. A particular manifestation is the jungle of assorted devices for reproducing limited human decision procedures. An attempt is made here to begin a systematic classification of the various devices for educating the computer to take over the decision-making functions of one or many human operators, both those that have been demonstrated feasible to date and those that are highly desirable for the future.

R. W. BEMER,
I.B.M. Corporation

■ It would be very presumptive to attempt to present an exhaustive survey of intelligence in even the most narrow and limited field, which the design and application of computer systems certainly is not, since it has the capability of representing the inherent universality of thought processes. This article is intended only to create an expandable framework for additions by others.

In this article, typical questions from such a checklist are amplified and explained so that some of the implications of such a check list can be appreciated by DATA Control readers with a detailed working knowledge of computer systems. Copies of the complete checklist are available to DATA Control readers upon request. (See box).

TYPICAL *PROCESSOR* QUESTIONS

Is all action taken on an exception basis, so that programs which use a minimum of facilities and least flexibility will be processed fastest?

A good example of this is the permitting of names of any number of characters. Suppose that the fixed word length of the computer will handle 5 alphabetic characters or special symbols. The programmer may then be cautioned that if he limits himself to names of 5 characters and less, only one word need be used for each and the processing will be much faster. If he used larger names, it will require longer tables and more lookup mechanism.

Are statements reserialized at each reprocessing to renew insert capacity?

Serial numbers have two purposes. The first is sequence checking of the statements in a source program. The second is the matching of corrections against the old source program, which is presumably on a medium which does not allow manual rearrangement. A common procedure is to either hand-serialize the original program, ignoring the lowest order position or have the first processing do this automatically if one is sure he will not disorder the program before processing. This lowest position is used for insertion of forgotten or additional statements, up to 9 if only numeric, and 35 if alphanumeric.

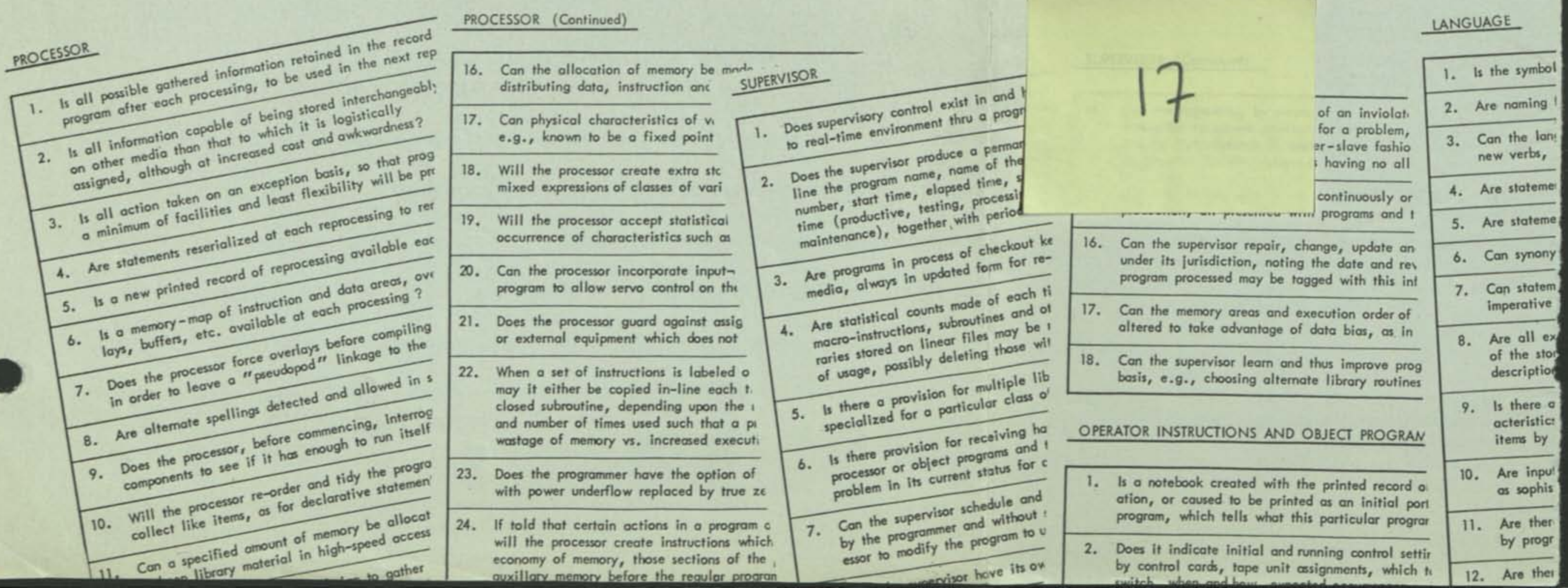
Since the processor can assign serial numbers automatically for the first processing, it is only reasonable to give it the further responsibility of doing the same thing every time, which reopens tight spots for more inserts and in general tidies up the program listing. Each page of the listing should have its own number, and lines should be in sequence on that page, starting from 1. This is extra inducement for the programmer to use the updated machine-produced listing as his only source for changes on further corrections.

Does the processor force overlays before compiling a full memory load, to leave a "pseudopod" linkage to the supervisory routine?

It is always best to leave a free area of high speed memory to control overlays. Without such a buffer, the processor will fill memory completely and have no means to call in another section when the program is inevitably expanded. With it, a minimum change in assembly is possible by trivial patching. Otherwise, the section of program which is displaced may have been referred to by other sections, all of which must therefore be reassembled.

Will the processor re-order and tidy the program at each processing to collect like items, as for declarative statements mixed with imperative?

It is an imposition upon the programmer to have to constantly re-



PROCESSOR

1. Is all possible gathered information retained in the record program after each processing, to be used in the next rep
2. Is all information capable of being stored interchangeabl on other media than that to which it is logistically assigned, although at increased cost and awkwardness?
3. Is all action taken on an exception basis, so that prog a minimum of facilities and least flexibility will be pr
4. Are statements reserialized at each reprocessing to rer
5. Is a new printed record of reprocessing available eac
6. Is a memory-map of instruction and data areas, ove lays, buffers, etc. available at each processing ?
7. Does the processor force overlays before compiling in order to leave a "pseudopod" linkage to the
8. Are alternate spellings detected and allowed in s
9. Does the processor, before commencing, interrog components to see if it has enough to run itself
10. Will the processor re-order and tidy the progra collect like items, as for declarative statemen
11. Can a specified amount of memory be allocat library material in high-speed access to gather

PROCESSOR (Continued)

16. Can the allocation of memory be made distributing data, instruction anc
17. Can physical characteristics of v e.g., known to be a fixed point
18. Will the processor create extra stc mixed expressions of classes of vari
19. Will the processor accept statistical occurrence of characteristics such as
20. Can the processor incorporate input- program to allow servo control on the
21. Does the processor guard against assig or external equipment which does not
22. When a set of instructions is labeled o may it either be copied in-line each t closed subroutine, depending upon the and number of times used such that a p wastage of memory vs. increased executi
23. Does the programmer have the option of with power underflow replaced by true ze
24. If told that certain actions in a program c will the processor create instructions which economy of memory, those sections of the auxillary memory before the regular program

SUPERVISOR

1. Does supervisory control exist in and to real-time environment thru a progr
2. Does the supervisor produce a perman line the program name, name of the number, start time, elapsed time, s time (productive, testing, processi maintenance), together with period
3. Are programs in process of checkout ke media, always in updated form for re-
4. Are statistical counts made of each ti macro-instructions, subroutines and ot raries stored on linear files may be with those wi of usage, possibly deleting those wit
5. Is there a provision for multiple lib specialized for a particular class o
6. Is there provision for receiving ha processor or object programs and problem in its current status for c
7. Can the supervisor schedule and by the programmer and without essor to modify the program to u
supervisor have its ow

17

of an inviolat
for a problem,
er-slave fashio
having no all
continuously or
programs and
16. Can the supervisor repair, change, update an under its jurisdiction, noting the date and re program processed may be tagged with this in
17. Can the memory areas and execution order of altered to take advantage of data bias, as in
18. Can the supervisor learn and thus improve prog basis, e.g., choosing alternate library routines

OPERATOR INSTRUCTIONS AND OBJECT PROGRAM

1. Is a notebook created with the printed record o ation, or caused to be printed as an initial port program, which tells what this particular prograr
2. Does it indicate initial and running control settir by control cards, tape unit assignments, which t switch, when and how, expected

LANGUAGE

1. Is the symbol
2. Are naming
3. Can the lan new verbs,
4. Are stateme
5. Are stateme
6. Can synony
7. Can statem imperative
8. Are all ex of the sto descriptio
9. Is there a acteristic items by
10. Are input as sophis
11. Are ther by progr
12. Are ther

order his source program so that non-acting definition or declarative statements are read before the statements requiring this information. The gathering of intelligence should be a series of sweeps thru the source program, and it is just as easy to extract this type of statement by means of the processor, which automatically puts them at the beginning of the program (where they would have been had the programmer been omniscient).

Will the processor create extra statements for or allow mixed expressions of classes of variables?

In general, the programmer should not have to constantly remember whether a variable is for the moment in fixed or floating point notation, single or double precision, rational or complex. The processor has access to conversion routines and should normally take care of this automatically.

Can the processor incorporate input-output interpreters in the object program to allow servo control on the basis of actual data characteristics?

Too much emphasis has been laid upon compiling the entire running program before operating. Particularly with the advent of simultaneous reading, calculating and writing, the balancing of these three functions to obtain optimum efficiency is most important. If one had to densely sprinkle the running program with interrogations of external equipment just to make decisions on the basis of its status, one would find that at least half of the memory was taken up with this function. A change of state of external equipment must interrupt the normal program sequence by a trap to an interrogatory program which assesses the need for rebalancing and will select the proper sections of program to do so for this condition. It is impossible for a compiler to choose an optimum mode of operations when the characteristics of the data are not known until running time and may change abruptly or periodically.

Computer Program Checklists Available

The accompanying article discusses typical questions from a checklist of intelligence for computer programming systems. The list consists of several series of questions concerning the processor, supervisor, operator instructions and object program, language and diagnostics. DATA Control readers may obtain complete copies of the actual checklist by writing:

DATA Control Editor
Automatic Control Magazine
430 Park Avenue
New York 22, N. Y.

When a set of instructions is labeled or called as a macro-instruction, may it either be copied in-line each time it is called or set up as a closed subroutine, depending upon the number of included instructions and number of times used such that a proper balance is obtained in wastage of memory vs. increased execution time for calling sequences?

There are various means of making this decision. Knowledge of how many times a routine will be executed dynamically for minimum execution time is difficult to come by, but economy of memory is possible thru static usage counts. Various weighted approximations may be used to give a simple formula for this determination. Take for example a routine which requires 4 instructions when compiled in line, using 3 extra instructions for linkage if compiled as a closed subroutine. If memory wastage were the only criteria and the routine were used in 10 places in the program, it would require 40 words in line and (10) (3) = 34 in a closed subroutine, which is better in this case although it takes more execution time. However, if used in only 3 places in the program, the in-line method uses slightly less memory, 12 to 13, and is considerably shorter in execution.

TYPICAL *SUPERVISOR* QUESTIONS

Does supervisory control exist in and have access to real-time environment through a programmable clock?

A programmable clock with a trapping interrupt feature can provide very useful decision data. Among the uses of such a clock are: (1.) Maintenance of a log of error frequency for statistical analysis by maintenance engineers. (2.) Determination of unstable or non-convergent iteration processes. (3.) Causing checkpoint procedures at selected time intervals. (4.) Allowing on-line operation in control systems. (5.) Making time studies of input-output balance. Keeping track of the real time required to execute various sections of program, for the processor to later reprogram for better balance and efficiency.

Can the supervisor schedule and select all components by names assigned by the programmer and, without stopping the computer, call upon the processor to modify the program to use alternate units when hardware fails?

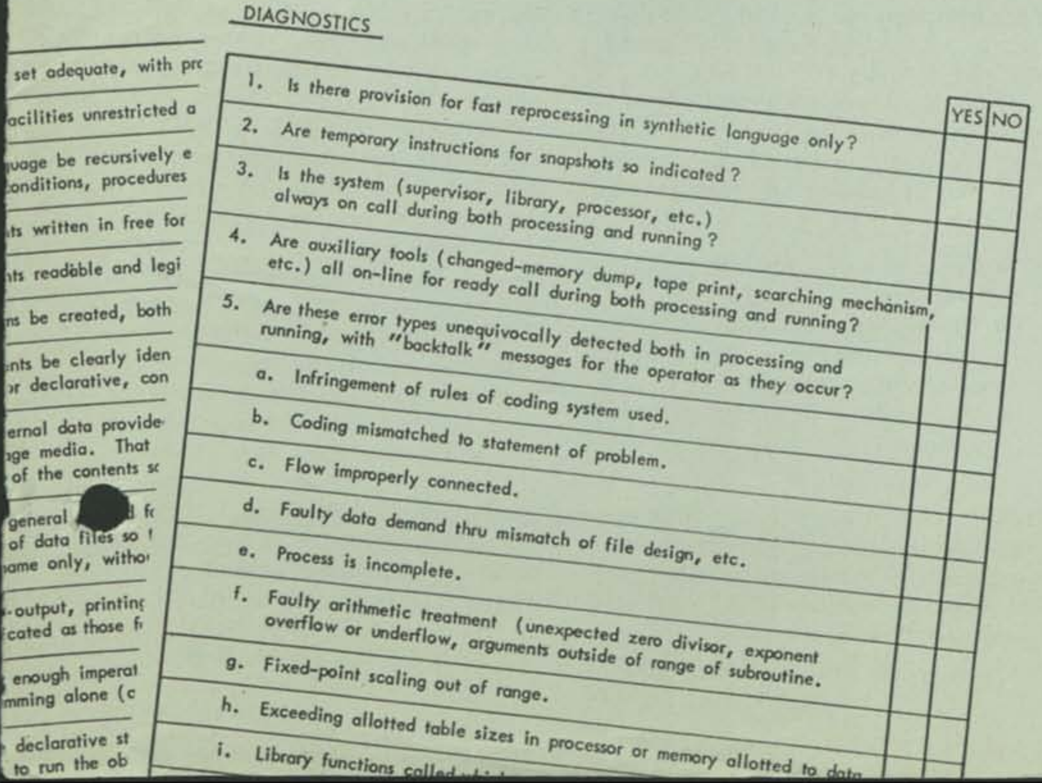
The input-output program should communicate with the supervisor to assign the correspondence between the logical (named) units, such as tapes, to which the programmer refers and the available units which the supervisor may use. The operators should also be informed of the units which are free for setting up the next job. This next job should read the tape labels on the new tapes and pass this information to the supervisor which, knowing now which tape contains what file, automatically reworks the program to call upon them properly. The physical unit number or designation thus makes no difference in the running of the job.

Is there provision to retrieve the processor to compile a section of program upon demand in the middle of object program execution?

In cases of many alternate procedures, it is wasteful of memory to compile machine instructions for all of these, particularly when only a few may be used in actuality. It is possible to simply compile traps to the supervisor for each of these alternatives. Then when such a program is actually needed, the trap to the supervisor calls in the processor and compiles an actual section of running program. It is less wasteful to keep such program alternates in low-speed memory in synthetic language form, than in expanded form in high-speed memory.

Can the supervisor schedule components for the most efficient use on a spectrum of problems?

Since the machine should never be allowed to stop, the supervisor(s) must be entrusted to manage the entire operation, scheduling automatically the various problems presented to it. Such an operation may be likened to that of a short order cook, the peripheral equipment to his order wheel and the customers to waiters placing orders on the wheel. Although the orders are placed in time sequence, the cook does not necessarily process in that order but rather tailors his operation to present and future loading of his facilities. In other words, the coffee, toast and scrambled eggs must all be done at the same time. The supervisor, upon completion of each job, should inspect all current orders, estimate their duration, note the components required and decide what to process next. It might well delay a long problem in order to do several quick problems in a row to



set adequate, with prc
acilities unrestricted a
uage be recursively e
onditions, procedures
ts written in free for
ts readable and legi
ns be created, both
nts be clearly iden
or declarative, con
ernal data provide
ge media. That
of the contents sc
general d fc
of data files so t
ame only, witho
-output, printing
cated as those fi
enough imperat
mming alone (c
declarative st
to run the ob

DIAGNOSTICS

	YES	NO
1. Is there provision for fast reprocessing in synthetic language only?		
2. Are temporary instructions for snapshots so indicated?		
3. Is the system (supervisor, library, processor, etc.) always on call during both processing and running?		
4. Are auxiliary tools (changed-memory dump, tape print, searching mechanism, etc.) all on-line for ready call during both processing and running?		
5. Are these error types unequivocally detected both in processing and running, with "backtalk" messages for the operator as they occur?		
a. Infringement of rules of coding system used.		
b. Coding mismatched to statement of problem.		
c. Flow improperly connected.		
d. Faulty data demand thru mismatch of file design, etc.		
e. Process is incomplete.		
f. Faulty arithmetic treatment (unexpected zero divisor, exponent overflow or underflow, arguments outside of range of subroutine.		
g. Fixed-point scaling out of range.		
h. Exceeding allotted table sizes in processor or memory allotted to data		
i. Library functions called		

19

January 26, 1960

TR 00.18000.705

AN EXTENDED CHARACTER SET STANDARD

by

R. W. Bemer and W. Buchholz

Product Development Laboratory, Data Systems Division
International Business Machines Corporation, Poughkeepsie, New York

AN EXTENDED CHARACTER SET STANDARD

by

R. W. Bemer and W. Buchholz

1. Introduction

Present IBM keying and printing equipment is designed to handle 48 or so characters and their codes. In order to guide the development of new equipment for keying and printing many more characters, an Extended Character Set (ECS) has been defined and is shown in Fig. 1. This set contains codes for 120 different characters, but there is room for later expansion to up to 256 characters including control characters. In addition, useful subsets have been defined which contain some but not all of these 120 characters and which use the same codes for the selected characters without translation.

It is intended that the design of future equipment conform to this standard code whenever the added versatility of an extended character set is desired. It is not expected that this code will obsolete the investment in equipment and methods on applications where a 48-character set is entirely adequate.

In selecting this character set, a great deal of thought has been given to satisfying a number of requirements. It proved impossible to satisfy all of them with a single ECS, but selecting a single ECS was considered to be an overriding requirement of the future. The need to communicate between data processing installations and the inevitable mixture of applications in a single installation make it more and more desirable to standardize. A standard character set is, of necessity, a compromise set. In any one situation it is always possible to define a better set, and there are obviously some applications demanding a highly specialized character set. Hence we do not consider this set to be ideal, but we do feel that it satisfies a great many of the more common requirements.

The main purpose of this report is to set down the requirements of an ECS as we see them, and to point out how they have or have not been met by this particular set.

2. Size of Set

Present IBM 48-character sets consist of

10 decimal digits,
26 capital letters,
11 special characters, and
blank.

Because a single set of 11 special characters is not sufficient, there exist several choices of special characters as "standard options".

Since this set is often represented by a 6-bit code, it is natural to try to extend this set to 63 characters and a blank, so as to exploit the full capacity of a 6-bit code. Although the extra 16 characters would be very useful, this step was thought not to reach far enough to justify the development of the new equipment which would be needed.

As a minimum, the new set should also include

26 lower-case letters,

the more important punctuation symbols found on all office typewriters, and

enough mathematical and logical symbols to satisfy the needs of programming languages such as ALGOL.

There is, of course, no definite upper limit on the number of characters. One could go to the Greek alphabet, various type fonts and sizes, etc., and reach numbers well into the thousands. As the set size increases, however, the cost and complexity of equipment goes up and the speed of printing goes down. The actual choice of 120 characters was purely a matter of judgment of what increment over existing sets would be sufficiently large to justify the departure from present codes without including many characters of only marginal value.

3. Subsets

Two subsets of 89 and 49 characters are shown in Figs. 2 and 3. The 89 character set (Fig. 2) is aimed at typewriters which, with 44 character keys, a case shift, and a space bar, can readily handle 89 characters.

This subset was considered important because input-output typewriters can already print 89 characters without modification, and 44-key keyboards are familiar to many people.

The 49-character subset (Fig. 3) is usable in a printer similar to the IBM 1403.* It represents the conventional set of "commercial" characters in a code which is compatible with the ECS. Thus, in a system equipped for the ECS it would still be possible to do high-volume printing efficiently on jobs (such as bill printing) where the extra characters may not be needed.

Other subsets are easily derived and may prove useful. For example, for purely numeric work, one may wish to construct a 13-character set consisting of the 10 digits and the symbols . and - together with a special blank.

4. Expansion of Set

Future expansion to a set larger than 120 may take place in two ways. One is to assign additional characters to presently unassigned 8-bit codes; allowance should be made for certain control codes which will be needed for communication and other devices and which are intended to occupy the high end of the code sequence. The second method is to define a shift character to "escape" to another character set. Thus, whenever the shift character is encountered, the next character (or group of characters) identifies a new character set, and subsequent codes are interpreted as belonging to that set. Another shift character in that set can be used to shift to a third set, which may again be the first set or a different set. Such additional sets would be defined only if and when there arise applications which require them.

5. Code

In choosing a code structure, many alternatives were considered. These varied in the size of the "byte" (i.e., the smallest number of information bits grouped together to represent a character) and in the number of bytes which may represent a single (printable) character. Among them were:

single 6-bit byte with shift codes interspersed,

double 6-bit byte = single 12-bit byte (Ref. 1),

single 8-bit byte,

single 12-bit byte for "standard" characters (punched card code) and two 12-bit bytes for other characters.

----

* Note that the IBM 1403 has available 49 characters including the blank, which is one more than the 48 characters on earlier printing equipment.

Some of these codes were attempts to remain, in some measure, compatible with earlier codes so as to take advantage of existing equipment. These attempts were abandoned, in spite of some rather ingenious proposals, because the advantages of partial compatibility were not enough to offset the disadvantages.

The 8-bit byte was chosen for the following reasons:

(a) The full capacity of 256 characters was considered to be sufficient for the great majority of applications for an ECS.

(b) Within the limits of this capacity, a single character is represented by a single byte so that the length of any particular record is not dependent on the coincidence of characters in that record.

(c) 8-bit bytes are reasonably economical of storage space.

(d) For purely numeric work, a decimal digit can be represented by only 4 bits and two such 4-bit bytes can be packed in an 8-bit byte. Although such packing of numeric data is not essential, it is a common practice to increase speed and storage efficiency. (The IBM 7070, for instance, uses an analogous scheme.) Strictly speaking, 4-bit bytes belong to a different code, but the simplicity of the 4 and 8 bit scheme, as compared to a 4 and 6 bit scheme, for example, leads to a simpler machine design and cleaner addressing logic.

(e) Byte sizes of 4 and 8 bits, being powers of 2, permit the computer designer to take advantage of powerful features of binary addressing and indexing to the bit level. (Ref. 2, 3).

In this report, the 8 bits of the code are numbered for identification from left to right as 0 (high-order bit) to 7 (low-order bit). "Bit 0" may be abbreviated to B_0, etc.

6. Parity Bit

For transmitting data, a ninth bit is attached to each byte for parity checking, and it is chosen so as to provide an odd number of one bits. Assuming a one bit to correspond to the presence of a signal, odd parity permits all 256 combinations of 8 bits to be transmitted and to be positively distinguished from the absence of any signal. The parity bit is identified here as B_p.

The parity bit is defined here for purposes of communication between devices and media using the ECS. It is not intended to exclude the possibilities of error correction or other checking techniques within a given device or on a given medium when appropriate.

7. Sequence

High-equal-low comparisons are an important aspect of data processing. Thus, in addition to defining a standard code for each character, one must also define a standard comparison ("collating") sequence. Obviously, the decimal digits must be sequenced from 0 to 9 in ascending order, and the alphabet from A to Z. Rather more arbitrary is the relationship between groups of characters, but the most prevalent convention for the 48 IBM "commercial" characters is, in order:

(Low)		Blank	
	11	Special Characters	. ⌑ & $ * - / , % # @
	26	Alphabet	A to Z
(High)	10	Digits	0 to 9

Fundamentally, the collating sequence of characters should conform to the natural sequence of the binary integers formed by the bits of that code. Thus 0000 0011 should collate below 0000 0100. Few existing codes have this property, and it is then necessary, in effect, to translate to a special internal code during alphanumeric comparisons. This takes extra equipment, extra time, or both. An important objective of the ECS was to obtain a usable collating sequence directly from the code without translation.

A second objective was to preserve the existing convention for the above 48 characters within the new code. This objective has only partly been achieved because of conflicts with other objectives.

The ECS provides the following collating sequence without any translation:

(Low)		Blank	
	43	Special Characters (see chart)	
	52	Alphabet	a A b B c C to z Z
	20	Digits	0 *0* 1 *1* to 9 *9*
	4	Special Characters	. : - ?
(High)		All Unassigned Character Codes	

Note that the lower and upper case letters collate in pairs in adjacent positions, following the convention established for directories of names. (There appeared to be no real precedent for the relative position within the pair. Telephone directories generally ignore the case shift for sequencing purposes, even though both upper and lower case letters appear within some names. This convention is not usable in general since each code must be considered unique.)

The difference between the ECS collating sequence and the earlier convention lies only in the special characters. Two of the previously available characters had to be placed at the high end and the remaining special characters do not fall in quite the same sequence with respect to each other. It was felt that the new sequence would be quite usable, and only rarely will it be necessary to re-sort a file in the transition to the ECS code. It is always possible to translate codes to obtain any other sequence, as has to be done with most existing codes.

8. Blank

The code 0000 0000 is a natural assignment for the blank (i.e., the non-print symbol which represents an empty character space). Not only should the blank collate below any printable character, but the absence of bits (other than the parity bit) corresponds to the absence of mechanical movement in a print mechanism.

Blank differs, however, from a "null" character, such as the all-ones code found on paper tape. Blank exists as a definite character occupying a definite position on a printed line, in a record, or in a field to be compared. A "null" may be used to delete an erroneous character and it would be completely dropped from a record at the earliest opportunity. Null, therefore, occupies no definite position in a collating sequence. A null has not been defined here, but it could be placed when needed among the control characters.

9. Typewriter Keyboard

Because the shift key on existing IBM 24 and 26 keypunches has been used to cause numbers to punch from otherwise alphabetic keys, it is necessary to establish a completely different convention when introducing lower-case letters. It was thought very desirable, therefore, to take advantage of the widespread familiarity with the typewriter keyboard and to capitalize on existing touch-typing skills as much as possible.

The common typewriter keyboard consists of up to 44 keys, and a separate case shift key. To preserve this relationship in the code, the 44 keys are represented by 6 bits of the code (B_1 to B_6) and the case shift by a separate bit (B_7). The case shift was assigned to the lowest-order bit so as to give the desired sequence between lower and upper case letters.

For ease of typing, the most commonly used characters should appear in the lower shift ($B_7=0$). This includes the decimal digits and, when both upper and lower case letters are used in ordinary text, the lower-case letters. (This convention is different from that for single-case typewriters presently used in many data processing systems; when no lower-case letters are available, the digits naturally appear in the same shift as the upper-case letters.) It is recognized that the typewriter keyboard is not the most efficient alphanumeric keyboard possible, but it would be unrealistic to expect a change in the foreseeable future. For purely numeric data, it is always possible to use a 10-key keyboard instead of, or in addition to, the typewriter keyboard.

It was not practical to retain the upper-lower case relationships of punctuation and other special characters commonly found on typewriter keyboards. There is no single convention anyway, and typists are already accustomed to finding differences in this area.

10. Decimal Digits

The most compact coding for decimal digits is a 4-bit code, and the natural choices for encoding 0 to 9 are the binary integers 0000 to 1001. As mentioned before, two such digits can be packed into an 8-bit byte; for example, the digits 28 in packed form could appear as

$$\underline{0010}\quad \underline{1000}$$

To represent decimal digits unambiguously in conjunction with other ECS characters, they must have a unique 8-bit representation. The obvious choice is to spread pairs of 4-bit bytes into separate 8-bit bytes and insert a 4-bit prefix ("zone"); for example, the digits 28 might be encoded as

$$z_a z_b z_c z_d\ \underline{0010} \qquad z_a z_b z_c z_d\ \underline{1000}$$

where the actual value of the zone bits z_i is immaterial so long as the prefix is the same for all digits.

This requirement conflicted with requirements for collating sequence and for the shift bit. As a result, the 4-bit byte is offset by one bit, and the actual code for 28 is

$$011\underline{0}\ \underline{010}0 \qquad 011\underline{1}\ \underline{000}0$$

This compromise retains the binary integer codes 0000 to 1001 in adjacent bit positions, but not in either of the two positions where they appear in the packed format.

The upper-case counterparts of the normal decimal digits are assigned to italicized decimal subscripts.

11. Adjacency

The 52 characters of the upper and lower case alphabets occupy 52 consecutive code positions without gaps. For the reasons given above, it was necessary to spread the 10 decimal digits into every other one of 20 adjacent code positions, but the remaining 10 positions are filled with logically related decimal subscripts. The alphabet and digit blocks are also contiguous. Empty positions for additional data and control characters are all consolidated at the high end of the code chart.

This grouping of related characters into solid blocks of codes, without empty slots that would sooner or later be filled with miscellaneous characters, assists greatly in the analysis and classification of data for editing purposes. Orderly expansion is provided for in advance.

12. Uniqueness

A basic principle underlying the choice of the ECS is to have only one code for each character and only one character for each code.

Much of the lack of standardization in existing character sets arises from the need for more characters than there are code positions available in the keying and printing equipment. Thus, in the existing 6-bit IBM character codes, the code 001100 may stand for any one of the characters @ or - or ' . The ECS was, instead, required to contain all of these characters with a unique code for each.

The opposite problem exists too. Thus - may be represented by either 100000 or 001100 in one of the existing 6-bit codes. Such an embarrassment of riches presents a logical problem when the two codes have in fact the same meaning and can be used interchangeably. No amount of comparing and sorting will bring like items together until one code is replaced by the other everywhere.

In going to a reasonably large ECS, it was necessary to resist a strong temptation to duplicate some characters in different code positions so as to provide equal facilities in non-overlapping subsets. Instead, every character was chosen to be typographically distinguishable even if the character stands by itself without context. Thus, for programming purposes, it is possible to represent any code, to which a character has been assigned, by its character even when the bit grouping does not have the ordinary meaning of that character (e.g., in operation codes).

In many instances, however, it is possible to find a substitute character which is close enough to a desired character to represent it in a more restricted subset or for other purposes. For example, = (equals) may stand for ← (is replaced by) in an 89-character subset. ⊙r again, if a hyphen is desired that collates below the alphabet, the symbol ⁓ (a modified tilde) is preferred to the more conventional - (minus).

A long-standing source of confusion has been the distinction between upper-case ⊙h (O) and Zero (0). Some groups have solved this by writing Zero as ∅ . Unfortunately, other groups have chosen to write ⊙h as Ø . Neither solution is typographically attractive. Instead, it is proposed to modify the upper-case ⊙h by a center dot and to write and print it as ⊙ whenever a distinction is desired.

Serifs are used to distinguish letters (I, l, V, etc.) from other characters (| , 1 , V, etc.) . It is suggested that the italicized subscripts be underlined when handwritten by themselves, e.g., $\underline{5}$.

13. Signs

The principle of uniqueness implies a separate 8-bit byte to represent a + or - sign. Keying and printing equipment also require separate sign characters. This practice is, of course, rather expensive in storage space, but it was considered superior to the ambiguity of present 6-bit codes where otherwise "unused" zone bits in numeric fields are used to encode signs. If the objective is to save space, one may as well abandon the alphanumeric code quite frankly and switch to a 4-bit decimal coding with a 4-bit sign digit or go to the even more compact binary radix.

14. Card Punching

After considering the possibility of a separate card code for the ECS characters, a code which has the conventional IBM card code as a subset (Ref. 1), it was concluded that it would be better to punch the ECS code directly into the card. This does not preclude also punching the conventional code (limited to 48 characters) in part of the card for use with conventional equipment. In this way, code translation is needed only wherever the conventional card code is used; if a non-ECS code were used, translation would be required for every column if advantage is to be taken of the ECS code in the rest of the system.

The punching convention is as follows:

Card Row	ECS Bit
12	---
11	---
0	---
1	B_p
2	B_0
3	B_1
4	B_2
5	B_3
6	B_4
7	B_5
8	B_6
9	B_7

In addition, both 12 and 11 holes are to be punched in column 1 of every card containing the ECS code, in addition to a regular ECS character, so as to distinguish an ECS card from cards punched with the conventional code. ECS punching always starts in column 1 and extends as far as desired; a control code "End" (0 1111 1110) has been defined to terminate the ECS code area. Conventional card code punching should be confined to the right end of cards identified with 12-11 punching in column 1.

Since the parity bit is also punched, the ECS area of a card contains a checkable code. Note that "blank" columns in the ECS area still have a hole in the B_p row. If only part of the card is to be punched, however, it is possible to leave the remaining columns on the right unpunched.

15. Acknowledgements

The Extended Character Set described here was developed jointly by E. G. Law, H. J. Smith, F. A. Williams, and the authors.

16. References

1. R. W. Bemer, "A Proposal for a Generalized Card Code for 256 Characters," Communications of the ACM; Vol. 2, No. 9, September 1959.

2. F. P. Brooks, Jr., G. A. Blaauw, W. Buchholz, "Processing Data in Bits and Pieces," IRE Transactions on Electronic Computers; EC-8, No. 2, June 1959.

3. W. Buchholz, "Fingers or Fists? (The Choice of Decimal or Binary Representation)," Communications of the ACM; Vol. 2, No. 12, December 1959.

120 CHARACTER SET

	BITS 0-1-2-3							
BITS 4-5-6-7	0000	0001	0010	0011	0100	0101	0110	0111
0000	BLANK	[	+	c	k	s	0	8
0001	±	⊃	&	C	K	S	*0*	*8*
0010	→	]	$	d	l	t	1	9
0011	{	°	(	D	L	T	*1*	*9*
0100	∧	←	*	e	m	u	2	.
0101	}	≡	)	E	M	U	*2*	:
0110	↑	¬	/	f	n	v	3	-
0111	≠	√	=	F	N	V	*3*	?
1000	∨	%	,	g	o	w	4	
1001	∀	\	;	G	Θ	W	*4*	
1010	↓	◇	'	h	p	x	5	
1011	‖	|	"	H	P	X	*5*	
1100	>	#	a	i	q	y	6	
1101	≧	!	A	I	Q	Y	*6*	
1110	<	@	b	j	r	z	7	
1111	≦	~	B	J	R	Z	*7*	

Fig. 1.

89 CHARACTER SET

BITS 4-5-6-7	BITS 0-1-2-3							
	0000	0001	0010	0011	0100	0101	0110	0111
0000	BLANK		+	c	k	s	0	8
0001			&	C	K	S	*0*	*8*
0010			$	d	l	t	1	9
0011			(	D	L	T	*1*	*9*
0100			*	e	m	u	2	.
0101			)	E	M	U	*2*	:
0110			/	f	n	v	3	–
0111			=	F	N	V	*3*	?
1000			,	g	o	w	4	
1001			;	G	⊙	W	*4*	
1010			'	h	p	x	5	
1011			"	H	P	X	*5*	
1100			a	i	q	y	6	
1101			A	I	Q	Y	*6*	
1110			b	j	r	z	7	
1111			B	J	R	Z	*7*	

Fig. 2.

49 CHARACTER SET

	BITS 0-1-2-3							
BITS 4-5-6-7	0000	0001	0010	0011	0100	0101	0110	0111
0000	BLANK						0	8
0001			&	C	K	S		
0010			$				1	9
0011				D	L	T		
0100			*				2	.
0101				E	M	U		
0110			/				3	-
0111				F	N	V		
1000		%	,				4	
1001				G	⊙	W		
1010		◇	'				5	
1011				H	P	X		
1100		#					6	
1101			A	I	Q	Y		
1110		@					7	
1111			B	J	R	Z		

Fig. 3.

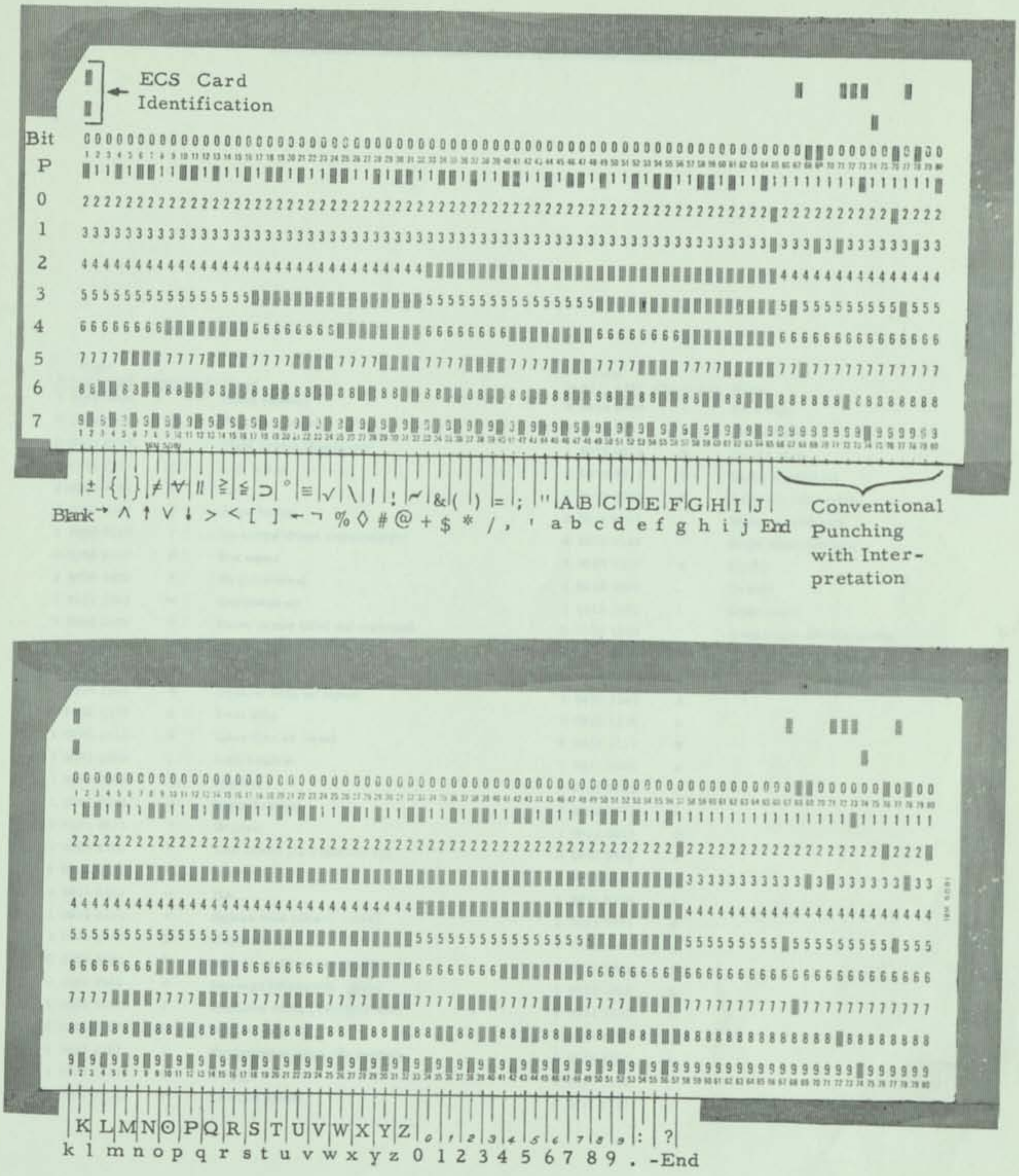

ECS Card Identification
Bit
P
0
1
2
3
4
5
6
7
± { } ≠ ∀ ‖ ≧ ≦ ⊃ ° ≡ √ \ | ! ~ & ( ) = ; " A B C D E F G H I J
Blank→ ∧ ↑ ∨ ↓ > < [ ] ← ¬ % ◊ # @ + $ * / , ' a b c d e f g h i j End
Conventional Punching with Interpretation
K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 : ?
k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9 . - End

Fig. 4. Cards Punched with Extended Character Code.

APPENDIX A

List of ECS Codes and Characters

Code P 0123 4567	Character	Name
1 0000 0000		Blank (Space)
0 0000 0001	±	Plus or minus
0 0000 0010	→	Right arrow (Replaces)
1 0000 0011	{	Left brace
0 0000 0100	∧	And
1 0000 0101	}	Right brace
1 0000 0110	↑	Up arrow (Start superscript)
0 0000 0111	≠	Not equal
0 0000 1000	∨	Or (inclusive)
1 0000 1001	⊻	Exclusive or
1 0000 1010	↓	Down arrow (End superscript)
0 0000 1011	‖	Double lines
1 0000 1100	>	Greater than
0 0000 1101	≥	Greater than or equal
0 0000 1110	<	Less than
1 0000 1111	≤	Less than or equal
0 0001 0000	[	Left bracket
1 0001 0001	⊃	Implies
1 0001 0010	]	Right bracket
0 0001 0011	°	Degree
1 0001 0100	←	Left arrow (Is replaced by)
0 0001 0101	≡	Identical
0 0001 0110	¬	Not
1 0001 0111	√	Square root (Check mark)
1 0001 1000	%	Percent sign
0 0001 1001	\	Left slant (Reverse divide)
0 0001 1010	◊	Lozenge (Diamond) (Note)
1 0001 1011	|	Absolute value (Vertical line)
0 0001 1100	#	Number sign
1 0001 1101	!	Exclamation point (Factorial)
1 0001 1110	@	At sign
0 0001 1111	~	Tilde (Hyphen)

Code P 0123 4567	Character	Name
0 0010 0000	+	Plus sign
1 0010 0001	&	Ampersand
1 0010 0010	$	Dollar sign
0 0010 0011	(	Left parenthesis
1 0010 0100	*	Asterisk (Multiply)
0 0010 0101	)	Right parenthesis
0 0010 0110	/	Right slant (Divide)
1 0010 0111	=	Equals
1 0010 1000	,	Comma
0 0010 1001	;	Semi-colon
0 0010 1010	'	Apostrophe (Single quote)
1 0010 1011	"	Ditto (Double quote)
0 0010 1100	a	
1 0010 1101	A	
1 0010 1110	b	
0 0010 1111	B	
1 0011 0000	c	
0 0011 0001	C	
0 0011 0010	d	
1 0011 0011	D	
0 0011 0100	e	
1 0011 0101	E	
1 0011 0110	f	
0 0011 0111	F	
0 0011 1000	g	
1 0011 1001	G	
1 0011 1010	h	
0 0011 1011	H	
1 0011 1100	i	
0 0011 1101	I	
0 0011 1110	j	
1 0011 1111	J	

Code P 0123 4567	Character	Name
0 0100 0000	k	
1 0100 0001	K	
1 0100 0010	l	
0 0100 0011	L	
1 0100 0100	m	
0 0100 0101	M	
0 0100 0110	n	
1 0100 0111	N	
1 0100 1000	o	
0 0100 1001	O	
0 0100 1010	p	
1 0100 1011	P	
0 0100 1100	q	
1 0100 1101	Q	
1 0100 1110	r	
0 0100 1111	R	
1 0101 0000	s	
0 0101 0001	S	
0 0101 0010	t	
1 0101 0011	T	
0 0101 0100	u	
1 0101 0101	U	
1 0101 0110	v	
0 0101 0111	V	
0 0101 1000	w	
1 0101 1001	W	
1 0101 1010	x	
0 0101 1011	X	
1 0101 1100	y	
0 0101 1101	Y	
0 0101 1110	z	
1 0101 1111	Z	

Code P 0123 4567	Character	Name
1 0110 0000	0	Zero
0 0110 0001	$_0$	Subscript zero
0 0110 0010	1	One
1 0110 0011	$_1$	Subscript one
0 0110 0100	2	Two
1 0110 0101	$_2$	Subscript two
1 0110 0110	3	Three
0 0110 0111	$_3$	Subscript three
0 0110 1000	4	Four
1 0110 1001	$_4$	Subscript four
1 0110 1010	5	Five
0 0110 1011	$_5$	Subscript five
1 0110 1100	6	Six
0 0110 1101	$_6$	Subscript six
0 0110 1110	7	Seven
1 0110 1111	$_7$	Subscript seven
0 0111 0000	8	Eight
1 0111 0001	$_8$	Subscript eight
1 0111 0010	9	Nine
0 0111 0011	$_9$	Subscript nine
1 0111 0100	.	Period (Point)
0 0111 0101	:	Colon
0 0111 0110	-	Minus sign
1 0111 0111	?	Question mark

NOTE: The character ⌗ has also been used.

APPENDIX B

6-Bit Character Codes in Current IBM Systems

The special characters shown in the attached chart are those of the "commercial" set (A). For "scientific" computing, character substitutions (sets F and H) are usually made for codes representing certain symbols in the 'A' set:

Commercial	Scientific	
A	F	H
&	+	+
⌑	)	)
%	(	(
#	=	=
@	-	'

Code	8 Ch. Paper Tape	305, 1620 P. T.	650 Mag. Tape	705; 704 M. T.	7070 Mag. Tape	1401	704 Internal	Code	8 Ch. Paper Tape	305, 1620 P. T.	650 Mag. Tape	705; 704 M. T.	7070 Mag. Tape	1401	704 Internal
00 0000	b	b		Sp		b	0	10 0000	-	-	-	-	-	-	-
00 0001	1	1	1	1	1	1	1	10 0001	J	J	J	J	J	J	J
00 0010	2	2	2	2	2	2	2	10 0010	K	K	K	K	K	K	K
00 0011	3	3	3	3	3	3	3	10 0011	L	L	L	L	L	L	L
00 0100	4	4	4	4	4	4	4	10 0100	M	M	M	M	M	M	M
00 0101	5	5	5	5	5	5	5	10 0101	N	N	N	N	N	N	N
00 0110	6	6	6	6	6	6	6	10 0110	O	O	O	O	O	O	O
00 0111	7	7	7	7	7	7	7	10 0111	P	P	P	P	P	P	P
00 1000	8	8	8	8	8	8	8	10 1000	Q	Q	Q	Q	Q	Q	Q
00 1001	9	9	9	9	9	9	9	10 1001	R	R	R	R	R	R	R
00 1010	PI1		0	0	0	0		10 1010	PI2			$\bar{0}$	$\bar{0}$	$\bar{0}$	$\bar{0}$
00 1011	#	#	#	#	#	#	#	10 1011	$	$	$	$	$	$	$
00 1100	@	@	@	@	@	@	@	10 1100	*	*	*	*	*	*	*
00 1101	PI7							10 1101	PI6						
00 1110	EC1							10 1110	CR						
00 1111	Cor		TM	TM	TM	TM		10 1111	Err				△	△	
01 0000	0	0	b	b	b	Sp	&	11 0000	&	&	&	&	&	&	b
01 0001	/	/	/	/	/	/	A	11 0001	A	A	A	A	A	A	/
01 0010	S	S	S	S	S	S	B	11 0010	B	B	B	B	B	B	S
01 0011	T	T	T	T	T	T	C	11 0011	C	C	C	C	C	C	T
01 0100	U	U	U	U	U	U	D	11 0100	D	D	D	D	D	D	U
01 0101	V	V	V	V	V	V	E	11 0101	E	E	E	E	E	E	V
01 0110	W	W	W	W	W	W	F	11 0110	F	F	F	F	F	F	W
01 0111	X	X	X	X	X	X	G	11 0111	G	G	G	G	G	G	X
01 1000	Y	Y	Y	Y	Y	Y	H	11 1000	H	H	H	H	H	H	Y
01 1001	Z	Z	Z	Z	Z	Z	I	11 1001	I	I	I	I	I	I	Z
01 1010	PI3		RM	RM	RM	RM	$\overset{+}{0}$	11 1010	SP1			$\overset{+}{0}$	$\overset{+}{0}$	$\overset{+}{0}$	
01 1011	,	,	,	,	,	,	.	11 1011	.	.	.	.	.	.	,
01 1100	%	%	%	%	%	%	⌑	11 1100	⌑	⌑	⌑	⌑	⌑	⌑	%
01 1101	PI4				WS			11 1101	PI5						
01 1110	Skip							11 1110	SP2						
01 1111	EC2			SM	SM			11 1111	TF			GM	GM	GM	

Note: b stands for blank.

6. Output media
7. Running time
8. Decimal places required for accuracy
9. Errors. Error stops or other stops are listed with reference to the appropriate section of the flow chart. The listing also tells what should be done in case of an error stop.

IV. Flow Chart—Problem-oriented rather than computer-oriented and using a standard set of symbols.

V. The Program Itself. This section consists of the following items:

a. Language and program steps: a listing of the complete program for at least one computer
b. Cross references from one program step to other steps not in sequence
c. Operating instructions including plugboard wiring diagrams where necessary

VI. Sample Problems. Included in the sample problems are the required input data and the desired output format showing the results of the problem.

VII. Notation. This is a list of the nomenclature used in the text and of the symbols in the source program listing.

VIII. Literature References.

IX. Illustrations. Drawings and other illustrative material, prepared in a manner to insure good reproduction, is included in this section.

Editor's Comment: While the ACM has never adopted publication standards, it now appears that *program publication standards* for the ACM may be desirable in the near future. In my opinion, not only is the interest in program interchange growing constantly, but computer-independent language development is bringing us closer to the day when widespread publication of truly catholic programs will be an important function of computer-oriented societies.

Mr. Kent's abridgment of the AIChE committee's report avoids machine-oriented details such as punched card formats; it outlines a procedure which is essentially machine-independent, yet can be used for machine-language publication.

The full report describes means by which program publication is announced and plans for distribution and for covering the cost of small-scale publication by the Society. H. S. B.

A Proposal for Character Code Compatibility

R. W. Bemer *I.B.M. Corporation, White Plains, N. Y.*

The emergence of a single standard from a welter of conflicting precedents depends upon two solutions:

1. selection or development of an adequate and logical standard,
2. phasing out (or peaceful coexistence with) the old varieties.

This paper deals with the latter problem and proposes the mechanics for a solution in the area of character codes, as represented by bit combinations.

It appears impossible to reconcile the many different codes in use on paper or magnetic tape such that a particular code could be the national or international standard. Because of the wide usage of these various codes they must be considered parallel standards subject to atrophy through adoption of a single superior code. A simple device that I call the "escape" character will allow as many important and graded standards as there are bit combinations for any number of tracks, although it is certainly not desirable to have more of these than absolutely necessary.

Given 7 character tracks (not feed, parity, or control tracks) there are 2^7 possible code combinations. Normally these are all assigned to specific characters or controls. I propose that *one* of these combinations, the *same* one for all standards, be reserved as an "escape" character. This is to be excluded from every such set of characters assigned.

Regarding the choice of this character, it is unwise to use a *null*, or absence of punches or bits. Furthermore, it is quite possible that the physical permutation of tracks on tape will not be in direct correspondence with the bit pattern of internal storage in a computer or data-processing device. The only code that avoids these difficulties is the completely punched combination, or all *ones* in the bit structure.

Let us make provision for this "escape" combination to interrupt normal decoding of a stream of characters. It will say, in effect, that "The next 7-bit combination is to be considered a numeric identifier of a particular standard." From then on, until interrupted by an "escape" character in *that* set, all combinational 7-bit characters will be interpreted according to that standard. Shifting from one standard to another is therefore *dynamic*. A great additional advantage of such a scheme is that many messages in several different codes may be adjoined in the stream of transmission. In hardware, the "escape" character can be made to interrupt to set relays or other switching devices to select one of a variety of readers or decoders.

1960 FEB

THIS COPY TO RW BEMER.

8.1.Turing,1982.1.2.1.2

COMPLAN®

COMPUTATION PLANNING, Inc.,● 7840 Aberdeen Road Bethesda, Md. 20814 ●(301) 654-1800

March 11, 1982

—— ACM TURING AWARD SUBCOMMITTEE MATTER ——

Mr. Charles W. Bachman, Vice President
Cullinane Database Systems, Inc.
400 Blue Hills Drive
Westwood, MA 02090

617-329-7700
800-225-9930

Dear Charlie:

This letter is written only to you and Pat Skelly, with a confirming copy to Bob Bemer to help keep me honest, regarding Pat's nomination of Bob as Turing Awardee. Some of the substance of this letter may be useful for your files at a later date.

On one subject —— perhaps on only one —— Bob and I are in complete agreement: our unbounded admiration for Bob.

In order to dispel any thought that I'm a Bemer buddy, I should remark that about the time of the incident described below Bob volunteered to replace me as ACM Standards Committee Chairman and CACM Standards Section Associate Editor if the Council would get me fired from both jobs . . . not for moral turpitude, alas, but for not doing enough in those jobs to satisfy Bob. Council failed to do so.

Bob chaired the then-ASA X3.1 Subcommittee on Standardization of Character Sets during and after my stint in the X3 chair, and was a principal creator of the ASCII set. I well remember nailing Bob once for an explanation of the "Escape Character" concept, which was new to me when Bob pushed it through his group. He convinced me no set of any length would meet all requirements even then, certainly not in the future, and that the Escape concept would work. His Subcommittee, competitors' employees all, grudgingly agreed. After some months of study, they became the strongest boosters of the idea.

Meanwhile, Bob's employer —— a large corporation —— seemed somewhat less than enthusiastic, if not downright reluctant, toward the whole ASCII set. It is a matter of public record that he changed employers about then. Several years later that company adopted ASCII.

The question has bugged me ever since: Is this an example of a solid company man placing his conviction about a public need above his well-developed instinct for bureaucratic self-preservation? There aren't too many in this business who, like you, seem to have displayed that bizarre character trait.

Having first raised, in Council, the idea of establishing a Turing Award, I have felt a fatherly concern about each selection.

I would feel comfortable if the award for this year goes to irritating, competent Bemer for the idea that will live after him.

Copy to: RWB, PGS

Yours truly,

Herbert S. Bright

Comment On COBOL

By Robert W. Bemer, IBM Corp., technical advisor to the Conference Committee on COBOL.

The IBM Corporation has been active in COBOL since its formation, recognizing the desirability of such a common business language. IBM desires to implement such a language for several of its computers when it is proved feasible and reasonably efficient for the user.

The extreme difficulties of developing such a language in a short period of time cannot be overemphasized. One has only to look at the gradual evolution of English to see what thousands of options have sprung up and then disappeared.

Although it is desirable to base a business language upon a natural language such as English (obviously for the convenience of the user), there are nevertheless certain restrictions of present day computers which make the variety in English undesirable. For example, we understand a man who speaks English even though he stutters, but this is not economical to expect the machine to decode stuttering. Primarily this is because the human mind operates very much in parallel, whereas the computer of today is largely serial—at least in its scanning. A person who misses the sense of a sentence has only to reread and check a few points. If the computer has to do anything more than a single progressive reading of a sentence, such as a see-saw inspection, the cost of translation becomes prohibitive. I know of a case where it took a computer over 11 hours to produce a machine program of somewhat more than 2,000 instructions.

When the language is formalized and the latitude of options removed, or to put it bluntly, if the user restrains himself with a little discipline, this same job should not take more than an hour.

Additional complications are, of course, caused by the fact that the language must be universal and roughly as effective for each of several decidedly different computers. Compromise is necessary! Eventually such compromise is well worth it, but this is a slow process. The goal of standardization in languages is very desirable, but it will not be served if the first product, i.e., the COBOL language, fails in the field. Before such a language can be hailed as a panacea, it must be subjected to extensive field tests.

IBM has put forth a major effort in this venture, supporting it with the services of many experts in computer languages. The experience gained with FORTRAN and the Commercial Translator has been freely given. Whether the goal of a common business language is achieveable without unduly compromising machine performance is not yet proven. In this situation, it is advisable to make haste slowly that we may not raise the hopes of our customers before it is justified.

COBOL–Commo

21

A UNIVERSAL computer language moved one step closer to reality with the announcement of COBOL (Common Business Oriented Language), a business language expected to be common to virtually all makes and models of electronic digital computers.

The new source language system will permit programmers to use English words, statements, sentences and paragraphs in communicating instructions to computer systems.

Official news of the COBOL development is anticipated momentarily from the project sponsors, the Executive Committee of the Conference on Data Systems, headed by C. A. Phillips, director of the data systems research staff, Department of Defense. The committee is a volunteer group of computer users from Government and industry and representatives of computer manufacturers.

Necessary violations

COBOL was written by the Conference's Short Range committee, directed by Joseph H. Wegstein of the National Bureau of Standards. This group, composed of technical personnel from three government agencies and six computer manufacturers, has worked continuously since June, 1959, to put the new language together.

One of the principles adopted in the development stage was that everything in the language would be correct English. This did not mean that everything which is correct English is meant to be part of the system or acceptable to a COBOL compiler. In some cases it was necessary to violate the principles of good English to allow inclusion of certain features which could not be handled by normal grammatical rules.

Language for Computers

MBA presents an exclusive account of the COBOL system, a move toward the ultimate development of a universal language for business computers

There are two basic elements to the COBOL system: (1) the Source Program, written in a common language, and (2) the Compiler, which translates this source program into an object program capable of running on a computer.

The source language is used to specify the solution of a business data processing problem. There are three elements—Procedure, Data and Environment—involved in this specification, and their names reflect the part of the over-all system which they describe. Procedure covers the set of procedures which determine how the data is to be processed. Data includes the description of the data being processed. Environment covers the description of the equipment being used in the processing. Each of the three elements are defined as a separate division of the system. The compiler's function is to integrate all of the divisions and produce the object program.

The Procedure division specifies the steps that the user wishes the computer to follow in order to produce the desired results. It allows the user to express his thoughts in English words, statements, sentences or paragraphs. Verb concepts denoting action and sentences describing procedures are basic. It is also possible to use logical situations and "if" clauses to provide alternative paths of action. The fact that the Procedure division is essentially machine-independent is one of the most important characteristics of the COBOL system. Another programmer, or any COBOL compiler, can easily understand and translate the information appearing in this division without regard to a particular computer.

The Data division uses "file" and "record descriptions" to describe the files of data that the object program is to manipulate or create and the individual logical records which comprise these files. Certain physical characteristics of the files are specifically not included—meaning that the Data division, to a certain extent, is also machine-independent.

A group of unique characters

The Environment division is that part of the source program which specifies the equipment being used. It contains descriptions of the computers to be used for compiling the source program and running the object program. Memory size, number of tape units, hardware switches and printers are among the many items that may be mentioned for a particular computer. The division has the ability to relate general program terminology to specific equipment. Those aspects of a file which relate directly to hardware are also described. Since this division deals entirely with the specifications of the equipment, it is entirely machine-dependent.

Thus, it can be seen that the amount of inter-machine compatibility throughout the COBOL system varies with the divisions and the effort taken to obtain such compatibility. The Procedure division requires virtually no effort to remain common across machines. In the case of the Data division, some care must be taken or a possible loss of efficiency may result. In the Environment division, all information is machine-dependent, therefore the compatibility is based on ease of understanding rather than direct transference.

COBOL uses 37 characters to make up the language's "words." The character set consists of the

The Short Range Committee, composed of representatives of six computer manufacturers and Government personnel, studied the strength and weaknesses of existing automatic business compilers and came up with COBOL system described here.

numbers 0 through 9, the 26 letters of the alphabet and a hyphen (or minus) sign. Seven characters are used for punctuation. These include the standard quotation marks, left and right parenthesis, space (defined as a character), period, comma and semicolon. Eight additional characters are used to define the operations involved in formulas and relations. Altogether there are 51 unique characters which are recognized by the COBOL system.

What's in a word

A word in COBOL language can be composed of not more than 30 characters. Types of words include nouns and verbs plus a special category of "reserve" words which includes "correctives," "noise words," and "key words." A COBOL noun is defined as a single word which is applicable to such elements as "Data Name," "Condition Name," "Procedure Name," "Literal Name" and "Special Register Name."

A Data Name is a word with at least one alphabetical character which designates any data specified in the data description. A Condition Name is given to a value which a field (called a conditional variable) may assume. For example, the field called "Title" is considered a conditional variable. The values which it may assume, and which are written and defined in the Record Description, are Analyst, Programmer and Coder. These Condition Names may be used in conditional expressions. As an example, if the field "TITLE" were defined as one character in legnth—and the actual values 1, 2 and 3 were assigned respectively to the Condition Names ANALYST, PROGRAMMER and CODER — the conditional expression "IF CODER THEN" would generate a test of the field "TITLE" against the value "3."

Procedure Names are applied either to paragraphs or to sections and accordingly are known as paragraph names or section names. A procedure may be named to permit one procedure in the language to refer to others, or it may be purely numeric.

A Literal is a noun which has a value identical to those characters represented by the noun. It may be numeric, alphabetic or alpha-numeric.

Special Register is a five-decimal digit field which has been assigned the name TALLY. Its primary use is to hold information produced by the EXAMINE verb. It may also be used to hold information produced elsewhere in a program.

Verbs are single words which appear in the Procedure division and designate action. Two types of action are allowed—object computer action by a special verb or compiler action denoted by a compiler directing verb.

Noise for improvement

Reserve words may be used for syntactical purposes and may not be used as nouns or verbs. Connectives are used to denote the presence of a qualifier or the presence of a subscript. Noise words are used to improve the readability of the language—but, the presence or absence of noise words does not affect the meaning of the statement. Within any division, any one or more of its noise words may be substituted for any other. Key words are required in certain formats. They are used to complete the meaning of verbs or entries and therefore must be present and correctly spelled.

Every name in a COBOL program must be unique—either because no other name has the identical spelling, or because the name exists within a hierarchy or names. The name can be made unique by mentioning several higher elements in

Continued on Page 37

Control Data Produces New 160 Desk-size Computer

An MBA Product Preview

THE DESK-SIZE MODEL 160, all-transistorized electronic computer, has been announced by Control Data Corp.

The 160 has an array of building blocks and a magnetic core memory like those used in the Control Data 1604 — the company's new large-scale system also announced recently. It computes in terms of microseconds and can execute 60,000 instructions in one second. It is said to handle data transmissions to and from input-output equipment at speeds of up to 65,000 characters per second.

The company has set the price of the 160 at $60,000, making it available to a wide range of users. Suggested applications include statistical and business data processing, data conversion, engineering and scientific calculations, data logging and data acquisition, industrial control and communications systems.

The 160 is a single-address computer with high-speed parallel mode of operation. Storage cycle time is 6.4 microseconds. Basic add time is 12.8 microseconds. Information read is available 2.2 microseconds after start of cycle. Average execution time is calculated at 15 microseconds per instruction. The computer uses a five megacyle logic.

The company points out that full advantage of the speed and versatility of the system can be realized through its repertoire of 62 instructions and complete programming package—which includes 22-, 33-, and 44-bit fixed point arithmetic, floating point, complex floating point, decimal, floating decimal, and an algebraic compiler. Addressing modes include: no address, direct address, indirect address, and relative address. Available input-output devices include a 350 character-per-second paper tape reader, 60 character-per-second paper tape punch, electric typewriter, up to eight magnetic tape handlers, card reader, card punch, and a line printer. Circle No. 3-17

COBOL

Continued from page 24

the hierarchy. These higher elements are called "qualifiers" when used in this way, and the process is called "qualification." Two types of qualification are allowed; prefixing (i.e., adjectival modification) and suffixing. In the first instance, the nouns must appear in descending order of hierarchy (i.e. with the name being qualified as the last and all others in order). In the second case the nouns must appear in ascending order of hierarchy with either of the words "OF" or "IN" separating them (the choice between the two words is based on readability — they are logically equivalent).

Dimensional arrays

Taking "President Election Year" as an example, the hierarchy of data given is such that neither the field "YEAR" nor the field "ELECTION" are unique spellings. That is, both fields appear elsewhere in the Record Description. To reference the "YEAR" field, PRESIDENT and ELECTION are used as qualifiers, either as nouns used adjectively in a prefix (PRESIDENT ELECTION YEAR), or preceded by the connective "of" for a suffix (YEAR OF ELECTION OF PRESIDENT).

When a list of items is defined in a program, reference may be made to any particular one by "subscripting." The list may not be referred to with subscripts. The name being subscripted is followed by the subscript which is identified either by following the key word "FOR" or by being surrounded by parenthesis. In certain situations, complex tables may be defined which require more than one quantity to locate an item. COBOL permits arrays containing up to three dimensions. The order of subscripts, from left to right, is major, intermediate and minor. For example, the premium rate of an insurance policy might depend upon the age, weight and the state of residence of the policyholder. The table would be classified as three dimensional and each valid subscript must be a series of three words. Paren-

Continued on next page

Continued from page 37

theses must be used in this case because the key word "FOR" might be ambiguous. The resulting instruction would read: MULTIPLY POLICY-VALUE BY RATE (AGE, WEIGHT, STATE).

COBOL procedures are expressed in a manner similar (but not identical) to normal English prose. The largest unit is a section, which is composed of paragraphs. The latter is made up of sentences which are generally grouped for the purpose of describing a unified idea. The sentences are composed of sequences of statements, which in turn are made up of groups of words—normally verbs and operands. COBOL makes available to the programmer several means of expressing logical situations through the use of the "conditional" procedures. These "conditionals" generally involve the key word "IF" followed by the condition to be examined, followed by the operations to be performed. The operations may vary, depending upon the truth or falsity of the conditions. For example: IF X EQUALS Y, MOVE A TO B; OTHERWISE IF C EQUALS D, MOVE A TO D AND ALSO PERFORM X THROUGH Y.

On the level

Under the COBOL concept, data to be processed falls into three categories—that which is contained in files and enters or leaves the internal memory of the computer from specified areas; that which is developed internally and placed into intermediate or working storage, and constants which are defined by the user. For purposes of processing, the contents of a file are divided into logical records. By definition, a logical record is any consecutive set of information. In an inventory transaction file, for example, a logical record could be defined as a single transaction, or as all consecutive transactions which pertain to the same stock item. Several logical records may occupy a block (i.e., physical record), or a logical record may extend across physical records. The logical record concept is not restricted to file data, but carries over into the definition of working storages and constants which may be grouped into logical entities and defined by a Record Description.

"Individuality and creativeness can still flourish freely within the framework of effective language standards."—C. A. Phillips, chairman of the Executive Committee of the Conference on Data Systems.

File Description entry contains information pertaining to the physical aspects of a file; the manner in which the data is recorded on the file, the volume of data in the file, the size of the logical and physical records, the names of the label records contained in the file, the names of the data records which comprise the file, and the keys on which the data records are sorted. The listing of data and label records in a File Description entry serves as a cross reference between the file and the records it contains. If the Record Description for these records is not found within the Data division of the problem description, it still can be automatically called from the COBOL library.

A Record Description consists of a set of entries, each of which defines the characteristics of a particular unit of data. Since COBOL Record Descriptions involve a hierarchal structure, an entry giving only the general characteristics may be followed by a set of subordinate entries which together redescribe the unit in more specific terms. The contents of an entry may vary considerably, depending upon whether or not it is followed by subordinate entries. A file of job tickets sorted according to division, department, employe number and day of the week is a good example of this. If the logical record has been defined as all consecutive data pertaining to a single employe, the following levels could be defined: (1) A weekly job record which consists of . . ., (2) Daily job ticket groupings which consist of . . ., (3) Job tickets which consist of . . ., and (4) The individual fields within the job tickets.

Within a COBOL Record Description, the programmer organizes and defines data according to its relative level by writing separate entries for each level and for each item of data within each level. The definition of a particular item of data consists of the entry written for that level plus all following entries which are of a lower level. The level, itself, is shown by a level number which is relative to the largest element of data within the Record Description. Level numbers start at 1—for records—and may go as high as 49, but it is not expected that any problem will require the full 49 levels of data.

Divided Divisions

The Environment Division is the one part of the COBOL system which must be rewritten each time a given problem is run on a different machine. It has been included in the system to provide a standard way of expressing the machine-dependent information which must be included as the part of every problem.

The division has been divided into two sections — Configuration and Input-Output. The Configuration section deals with the over-all specifications of computers and is divided into three paragraphs: the Source-Computer, which defines the computer on which the COBOL Compiler is to be run; the Object-Computer, which defines the computer on which the program produced by the COBOL Compiler is to be run, and Special Names, which relate the actual names of the hardware used by the program to the names used in the program.

The Input-Output section deals with the definition of the external media and that information that will create the most efficient transmission and handling of data between the media and object program. The section is divided into two paragraphs: the I-O Control, which defines special input-output techniques, rerun, and multiple file tapes; and File-Control, which names and associates the files with the external media.

In the beginning

The COBOL Library contains three types of entries, corresponding to the three divisions of the COBOL system. Information describing machine configurations is retrievable through the use of the Copy in the Environment division. File and record descriptions are retrievable through the use of the Copy in the Data division. Procedure statements—commonly called subroutines—are retrievable through the use of the verb INCLUDE in the Procedure division. Each division is capable of obtaining material pertaining only to itself. The physical makeup of the COBOL library, as well as the maintenance and handling, are left to the individual implementor. The calling of library material produces the same effect as if the programmer had written the material in his source program.

COBOL had its beginning at a conference held at the University of Pennsylvania Computing Center on April 8, 1959. The meeting brought together a group representing users, manufacturers and universities to discuss the problem of developing a common business language. The group, headed by Phillips, observed the recent development of languages for automatic programming, such as Sperry-Rand's FLOWMATIC, IBM's COMTRAN, and AIMCO, developed jointly by the Air Material Command and Sperry-Rand. The conclusion was that it might be feasible to develop specifications for a problem-oriented but machine-independent common language for business problems. The Department of Defense, as an appropriate agency with a major interest in the field, was asked to undertake the project.

Action in the Pentagon

On May 28, 1959, a two-day meeting was called in the Pentagon by Phillips to discuss the organization of the project. The concept of three committees, Short Range, Intermediate Range, and Long Range, was agreed upon with appropriate time schedules. The Short Range Committee was composed of six manufacturers, Government representatives and the chairman, Mr. Wegstein. Its task—to accomplish a fact-finding study of the strength and weaknesses of existing automatic business compilers and develop an improved system. Members of the group include: Col. Alfred Asch, Capt. Erwin Vernon and Duane Hedges of the Air Material Command-USAF; Robert S. Barton, William Logan and Mrs. Mary K. Hawes of Burroughs Corp. (Mrs. Hawes is now with RCA); Howard Bromberg, Ben F. Cheydleur, Norman Discount, Karl Kozarsky, Rex McWilliams and Gerald Rosenkrantz of Radio Corp. of America; William Carter, Charles Gaudette and Miss Sue Knapp of Minneapolis-Honeywell (Mr. Carter is now with IBM); Miss Deborah Davidson, Vernon Reeves and Miss Jean E. Sammet of Sylvania Electric Products, Inc.; William Finley, Dan Goldstein and Edward F. Sommers of Sperry-Rand; Roy Goldfinger, William Selden and Miss Gertrude Tierney of IBM; Mrs. Frances E. Holberton and Mrs. Norah Taylor of David Taylor Model Basin, USN, and Roy Nutt of the Computer Science Corp.

The Intermediate Range group will take the COBOL package and begin to modify and refine it within a time schedule ending sometime in 1961. Chairman of the group is A. E. Smith of the Navy Department.

The final phase of the program is the responsibility of the Long Range committee. This group will explore the fundamentals and philosophies of all machine language, regardless of its use on scientific or business data problems. The objective is to develop a "super" language which might supersede all existing scientific and business language systems. Such an accomplishment would be the ultimate—the Universal Computer Language. A special subcommittee, consisting of Robert Curry, vice president and comptroller of Southern Railway; Howard Engstrom, vice president of Sperry-Rand, and John McPherson, vice president of IBM, is directing this effort.

Trial of a concept

Assisting the Conference Committee as technical advisors in all phases of the COBOL project are Dr. Grace Hopper of Sperry-Rand, and Robert W. Bemer of IBM.

As the Short Range committee report points out, the COBOL system is the first large-scale effort at writing business data processing problems for many computers in one language. As such, it will undergo the trials of any new concept. Improvements and additions will be made by the committee which has stated that it is making "every effort to insure that improvements and corrections will be made in an orderly fashion." Proper provisions have been taken to avoid invalidating existing users' investments in programming.

The COBOL system further marks a major move toward complete computer compatibility. Other benefits include a reduction in the time requirements and costs of programming. The program has the complete support of the computer industry, and manufacturers have agreed to implement the language with compilers or processors to translate COBOL to the language of their particular machines. They recognize, Chairman Phillips pointed out, that individuality and creativeness can still flourish freely within the framework of effective language standards. ■

Reprinted from the Communications of the Association for Computing Machinery
Volume 3, Number 10, October 1960
Made in U.S.A.

Do It by the Numbers—Digital Shorthand*

R. W. Bemer, *International Business Machines Corporation, White Plains, New York*

Abstract. Present communications systems transmit single characters in groups of coded pulses between simple terminal equipments. Since English words form only a sparse set of all possible alphabetic combinations, present methods are inefficient when computer systems are substituted for these terminals. Using numeric representations of entire words or common phrases (rather than character-by-character representations) requires approximately *one-third* of present transmission time. This saving is reflected in overall costs. Other benefits accrue in code and language translation schemes. Provision is made for transmission of purely numeric and/or binary streams, and for single character-transmission of non-dictionary words such as the names of people or places.

General Principles

Precedent may be found in the story of the comedians' club that sat around and laughed when a member said "38." In this case the entire story is represented by that single number. One working example is that of standard Western Union messages such as birthday greetings. Not everyone realizes that the entire message is not transmitted, only its number; this tells the receiver what verbal message to type out on a form. Another example is that of telephone numbers. A name and address may be transmitted in a more compact fashion by merely sending the number. The receiver, equipped with the same phone book ordered on number rather than name, can simply decode.

Overstandardization at the message level will not work generally for the infinite variety met in practical transmission. The single word, delimited by blanks, is the efficient denominator. An example of this is the book code that children use for ciphers. Here the page number, line number and nth position on the line define a specific word. These three numbers may be compressed to a single number by using fixed subfields. Thus, 0312806 would indicate the 6th word of the 28th line on the 31st page. A related method would be to number all the words in the dictionary sequentially starting with 1.

Ground Rules

English unabridged dictionaries contain less than 600,000 individual entries. The average speaking vocabulary is from 1000 to 2000 words, the average writing vocabulary from 6000 to 8000. A college graduate may have from 7500 to 10000 words to use. It has been said that a person with a 3000-word vocabulary can understand 95 percent of general speech.

Since $2^{22} \sim 4{,}000{,}000$, it seems that about 22 bits should be capable of representing any word in the language, with perhaps enough freedom and overcapacity to be informational if desired. That is, they may identify words as to tense, plurals, nouns, or verbs, etc.

Examples given will use the six-bit representation for letters proposed in a recent draft standard issued by the Electronic Industries Association [1].

Compression Method

In early developments, Shannon [2] represented all words by an invariant or constant number of bits. For reasons of economy this is not practical. The average length of an English word is usually taken as 5 letters, plus a delimiting blank. A minimum of 5 bits (yielding 32 combinations) is necessary to represent a character singly. Thus the average number of bits required to represent a word in this mode is 30. Reduction to 22 bits is not impressive.

Therefore, statistical frequency of usage may be used to provide representations of a variable number of bits. The problem then is how to decode the bit stream at the receiving end. This may be done by

(1) advance knowledge of a constant byte size,

(2) termination by recognition of a fixed bit pattern normally excluded from the code,

(3) self-definition, where the first n bits of a group indicate its length,

(4) termination by inspection of every nth bit, or bits in a single track.

Method 2 is due to Brillouin [3]. Each representation is distinguished by at least two consecutive 0 bits followed by a 1 bit. The stream of Figure 1 is decoded as an example.

$$\underbrace{1\,0\,1\,1\,1}\,0\,0\,\underbrace{1\,0\,1\,1\,0\,1\,1\,1\,0\,1\,0\,0}\,0\,0\,\underbrace{1\,0\,0\,1\,0\,1\,0}\,0\,0\,1$$

Fig. 1

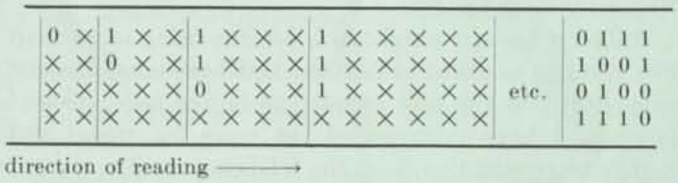
This method has the advantage of self-repair after a transmission error (except for the word in which the error occurred). It is not suitable for computer transformation of large dictionaries because of the exclusion of all address combinations with two consecutive zeros. It can be seen from Table 3 that it takes a storage of more than a half-million words to handle a vocabulary of 28,635 words, at an efficiency of only .055.

Self-defining (like a measuring worm), Method 3 is

* Presented at the meeting of the Association, August 23–26, 1960.

suitable to computer decoding. As an example, assume that transmission is to be in parallel on four channels (or on single wire in identifiable groups of four). Figure 2 shows how the leading bit(s) specify the length of a single word representation.

Leading bit(s)	No. of bytes (groups)	Working bits	Number of words accommodated	
0	2	7	128	(2^7)
10	3	10	896	$(2^{10} - 2^7)$
110	4	13	7,168	$(2^{13} - 2^{10})$
111	6	21	2,088,960	$(2^{21} - 2^{13})$



0 ×	1 × ×	1 × × ×	1 × × × × ×		0 1 1 1
× ×	0 × ×	1 × × ×	1 × × × × ×		1 0 0 1
× ×	× × ×	0 × × ×	1 × × × × ×	etc.	0 1 0 0
× ×	× × ×	× × × ×	× × × × × ×		1 1 1 0

direction of reading →

Fig. 2

The specific example at the right in Figure 2 means that the first word is decoded from the octal number 133 (or 000000000000001011011, in the full 21-bit address). There are many different methods of encoding for various byte sizes. Some are shown in Table 1. The example of Figure 2 is of Type B. The percent-usage figures are taken from Dewey's frequency study of 100,000 words [4]. The average number of bits per word may be reduced perhaps slightly from these figures by optimum adjustment to English frequency to the closest bit, rather than by the closest byte. However, this slight reduction is not warranted by the extra hardware and processing time. Note that the control patterns may be inverted or reassigned with exactly the same effect. Arbitrary change in the decoding rules is convenient for encrypting messages.

Correct positioning may be maintained for Method 3 by

(1) using intervening pulses of different length, as in Teletype,

(2) inserting synchronizing groups of all ones (1111 1111 1111) which have been excluded by the computer from the legitimate numbers sent,

(3) checking for reasonableness of message through statistical methods,

(4) guaranteeing that synchronization is never lost through self-checking methods (addition of parity bits, error-detecting and correcting codes, etc.).

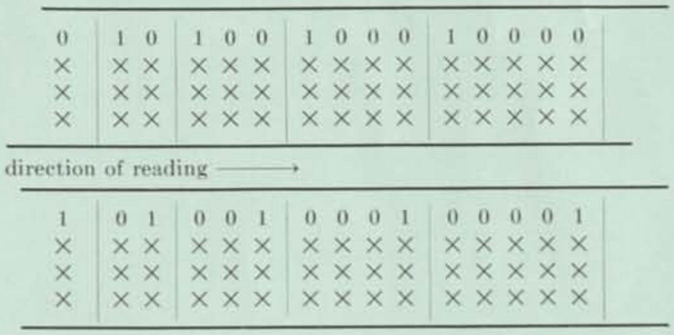
If an out-of-phase condition is likely, the proper receiving technique is to use a buffer area so the message can be re-interpreted. The amount of saving in this method will even allow the entire message to be sent twice, as an extreme measure. If the situation becomes intolerable in actual practice, Method 4 may be employed. Both Methods 3 and 4 have full storage utilization, as opposed to Brillouin's Method 2.

In Method 4, a single track is reserved for a wordmark. This wordmark can delimit in either of two ways, as shown in figure 3 (see p. 532).

At first appearance, this does not achieve the efficiency of Method 3. However, for all cases where the minimum number of bytes is two (i.e., Types A, B, D, H, J, K, M) the position adjacent to the initiating or terminal one might be used for information. This is because two one-bits in succession in the wordmark track constitutes an illegal condition.

TABLE 1. METHOD 3

Type	Bits per byte	First bits	No. of bytes	Working bits	Number of words accommodated	Percent usage (est.)	Percent times bytes	Bytes per word	Bits per word
A	4	0	2	7	128	57	114	2.69	10.76
		10	3	10	896	21	63		
		110	4	13	7,168	19	76		
		1110	5	16	57,344	2	10		
		1111	6	20	983,040	1	6		
B	4	0	2	7	128	57	114	2.71	10.84
		10	3	10	896	21	63		
		110	4	13	7,168	19	76		
		111	6	21	2,088,960	3	18		
C	5	0	1	4	16	28	28	2.19	10.95
		10	2	8	240	36	72		
		110	3	12	3,840	26	78		
		1110	4	16	61,440	9	36		
		1111	5	21	2,031,616	1	5		
D	4	00	2	6	64	48	96	2.76	11.04
		01	3	10	960	30	90		
		10	4	14	15,360	21	84		
		11	6	22	4,177,920	1	6		
E	6	0	1	5	32	38	38	1.85	11.10
		10	2	10	992	40	80		
		110	3	15	31,744	21	63		
		111	4	21	2,064,384	1	4		
F	5	0	1	4	16	28	28	2.28	11.40
		10	2	8	240	36	72		
		110	3	12	3,840	26	78		
		111	5	22	4,190,208	10	50		
G	6	00	1	4	16	28	28	1.95	11.70
		01	2	10	1,008	50	100		
		10	3	16	64,512	21	63		
		11	4	22	4,128,768	1	4		
H	5	0	2	9	512	71	142	2.34	11.70
		10	3	13	7,680	25	75		
		110	4	17	122,880	3	12		
		111	5	22	4,063,232	1	5		
J	5	0	2	9	512	71	142	2.37	11.85
		10	3	13	7,680	25	75		
		11	5	23	8,380,416	4	20		
K	5	00	2	8	256	64	128	2.41	12.05
		01	3	13	7,936	32	96		
		10	4	18	253,952	3	12		
		11	5	23	8,126,464	1	5		
L	6	0	1	5	32	38	38	2.06	12.72
		10	2	10	992	40	80		
		11	4	22	4,193,280	22	88		
M	6	0	2	11	2,048	84	168	2.17	13.02
		10	3	16	63,488	15	45		
		11	4	22	4,128,768	1	4		



```
0    1 0    1 0 0    1 0 0 0    1 0 0 0 0
×    × ×    × × ×    × × × ×    × × × × ×
×    × ×    × × ×    × × × ×    × × × × ×
×    × ×    × × ×    × × × ×    × × × × ×

direction of reading ———→

1    0 1    0 0 1    0 0 0 1    0 0 0 0 1
×    × ×    × × ×    × × × ×    × × × × ×
×    × ×    × × ×    × × × ×    × × × × ×
×    × ×    × × ×    × × × ×    × × × × ×
```

FIG. 3

A third method may deserve study, that of using alternate ones and zeros in the wordmark track, changing at the start of each new word. The advantage of using one extra bit for information is lost in this scheme. Table 2 shows corresponding efficiency in the wordmark mode. The corresponding efficiencies in bits per word are virtually the same as those of Table 1.

TABLE 2. METHOD 4

Method 3 Type	Bits per byte	Word mark bits	No. of bytes	Working bits
A	4	×1	2	7
		×01	3	10
		×001	4	13
		×0001	5	16
		×00001	6	19
B	4	×1	2	7
		×01	3	10
		×001	4	13
		×00001	6	19
C	5	1	1	4
		01	2	8
		001	3	12
		0001	4	16
		00001	5	20
E	6	1	1	5
		01	2	10
		001	3	15
		0001	4	20

For transmission on parallel wires, Method 4 is superior to Method 3 when errors occur.

It is also better for single wire transmission provided synchronization can be maintained regardless of error in any bit.

Code Efficiency

Brillouin [3] states that his code yields about 12 bits per word, very close to the theoretical lower limit that Shannon [2] believed to be 11.82. It is obvious from Table 1 that Types A through H are all better than Shannon's limit. Having duplicated Brillouin's work with the approximation to word frequency that he used (and also the Dewey frequencies), I get 10.12 bits per word with the frequency approximation that he used and 9.83 bits per word with the Dewey frequencies that Shannon used. The latter figure leads me to believe that Brillouin may have unknowingly penalized his scheme by 2 bits.

Although Baudot code is nominally 5 bits per character, the effective average is probably closer to 6 because of the extra shift characters and terminal blanks required to space words. Fieldata code does not use shift characters but does require terminal blanks and at least one extra bit, so the average is nearly 7 bits per character. Thus the scheme outlined in this paper will save between 60 and 65 percent over Baudot transmission and nearly 70 percent over Fieldata.

Brillouin has not published a scheme using more than two zeros as terminal indicators, and has stated that he believes it (the two-zero scheme) to be the most efficient possible. I have investigated the cases for three- and four-zeros termination, with the following results:

Number of zeros	Bits per word (Shannon)	Bits per word (Dewey frequency)
2	10.12	9.83
3	10.31	10.10
4	11.09	10.87

These figures are for a vocabulary of about 12,000 words. Table 3 shows that in this range the address efficiency of the three-zero terminator is about five times as good as that for two zeros. Thus it is actually much superior for practical computer operation. The reason an extra zero

TABLE 3.
BRILLOUIN CODES OF THE FORM 1 ×××× .00...

n = number of digits to left of the decimal point
Z = number of zeros in the terminator
U = number of usable combinations in each group

$U_n = U_{n-1} + U_{n-2} + \qquad 1 \qquad (\text{for } Z = 2)$

$U_n = U_{n-1} + U_{n-2} + U_{n-3} + \qquad 1 \qquad (\text{for } Z = 3)$

$U_n = U_{n-1} + U_{n-2} + U_{n-3} + U_{n-4} + 1 \qquad (\text{for } Z = 4)$

$$\text{Address efficiency} = \frac{\Sigma U}{2^n}$$

n	2^n	Z = 2		Z = 3		Z = 4		Address Efficiency		
		U	ΣU	U	ΣU	U	ΣU	Z = 2	Z = 3	Z = 4
1	2	1	1	1	1	1	1	.500	.500	.500
2	4	2	3	2	3	2	3	.750	.750	.750
3	8	4	7	4	7	4	7	.875	.875	.875
4	16	7	14	8	15	8	15	.875	.938	.938
5	32	12	26	15	30	16	31	.813	.938	.969
6	64	20	46	28	58	31	62	.719	.906	.969
7	128	33	79	52	110	60	122	.617	.859	.953
8	256	54	133	96	206	116	238	.520	.805	.930
9	512	88	221	177	383	224	462	.432	.748	.902
10	1024	143	364	326	709	432	894	.355	.692	.873
11	2048	232	596	600	1309	833	1727	.291	.639	.843
12	4096	376	972	1104	2413	1606	3333	.237	.589	.814
13	8192	609	1581	2031	4444	3096	6429	.193	.542	.785
14	16384	986	2567	3736	8180	5968	12397	.157	.499	.757
15	32768	1596	4163	6872	15052	11504	23901	.127	.459	.729
16	65536	2583	6746	12640	27692	22175	46076	.103	.423	.703
17	131072	4180	10926	23249	50941	42744	88820	.083	.389	.678
18	262144	6764	17690	42762	93703	82392	171212	.067	.357	.653
19	524288	10945	28635	78652	172355	158816	330028	.055	.329	.630
20	1048576	17710	46345	144664	317019	306128	636156	.045	.292	.607

does not add a full bit per word is that a higher proportion of the frequently used words may now be assigned to a denser set of addresses. The remaining choice between Brillouin's method and Methods 3 and 4 is now made as follows:

	Brillouin 00	Brillouin 000	Method 3,4	Baudot (Teletype)
Bits per word	9.83	10.10	10.62*	30.
Address efficiency	.06	.35	1.00	—

* adjusted for corresponding vocabulary size

Transformation Methods

A stored-program computer or a device with associative memory can transform a string of characters (each represented by a binary number) into a single compact and unique representation. An existing example of this statement is the conversion from binary coded decimal to pure binary numbers. The converse transformation at the receiving end requires only a simple address lookup to find the word or symbol to be printed.

The input typing or keying device produces B bits per character or letter. B is greater than or equal to 6 in order to handle at least 26 alphabetic characters, 10 digits and other necessary characters such as punctuation. A group of letters is deliminted by a blank, hyphen or other delimiter. Figure 4 shows the bit stream produced in the EIA code for the word PRIME. The blank delimits a **string** $(C_1, C_2, C_3, \cdots, C_N)$, N being the number of letters in the group, normally a word. This string is a number R_1. In Figure 4, $R_1 = 4547364232_{\text{octal}}$. The blank triggers the unloading of the buffer to a unit or program which transforms R_1 to a number R_t, which is the compressed representation transmitted on the communications line.

As the set of English words is sparse, the set of R_1 is also sparse. Transmission efficiency increases as the set of R_t is denser. R_t is also the address used at the receiving end of the line, $(R_t) = R_1$, which may either activate a character printing device or be retransmitted in decompressed form.

R_t is chosen to increase monotonically as frequency of the use of a word decreases. Thus the most frequently used words have lowest values of R_t and may thus have the leading zero bits truncated in the variable length mode. The program has an optional tally register associated with each word. Actual usage will generate practical frequencies which may be used for reassignment of the R_t values.

R_t may represent more than one word. After initial R_t's are formed, one per word, this compressed string may be inspected by matching pairs against a list of pairs which have high enough usage frequency to warrant condensation into a single R_t. This is recursive and any number of words may be represented by a single R_t. The only requirement is for the preceding R_t to remain in a buffer for matching.

Table Lookup Method

The length of a word is expressible in 5 bits. ($N_{\max}$ in English $= 28 < 2^5$ for the word antidisestablishmentarianism.) The computer storage is arranged to contain the operating program, a master table of N, C_1 and subordinate tables corresponding to all these values. For each word, N and C_1 are adjoined. (N, C_1) is found in the master table and is the address of the start of the proper table. In Figure 4 the value of N,C_1 is 0545_{octal}. (0545) = starting address for table of all five-letter words starting with P. $(N, C_1 + 1)$ = ending address of table + 1. Between these limits, a binary search finds a match to the value of R_1. Associated with R_1 in the table is the corresponding R_t.

This method does not make use of frequency information. It may be desired to place the tables randomly in storage. In this case the master table must be doubled in size. Adjoin the 5 bits of N, 6 bits of C_1, and a final low-order bit which indicates the starting address by 0, the ending address by 1. Finer grouping may be had at increased cost by using the concatenation of N, C_1, C_2.

Chaining Method

The entire number R_1 is utilized directly to find the corresponding R_t. A number M is chosen such that 2^M is convenient to storage size and related to vocabulary size for optimum conversion speed. For present storage sizes, M may vary from 10 to 14. The address R_1 modulo M has the contents:

$$R_1,\ R_t,\ \text{chain address}$$

The set of numbers $R_m = R_1$ modulo M will have duplicates and will not be dense, although denser than the

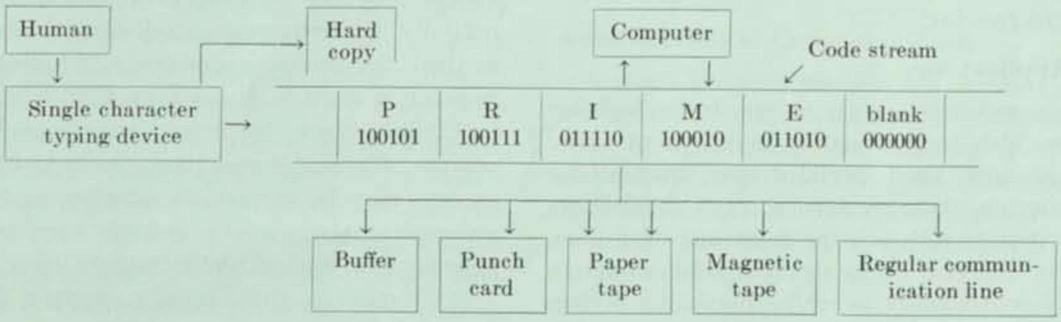


FIG. 4

set of R_1. When there are duplicates, the contents of R_m will not contain the proper R_1 except for one word. If not, the contents of the chain address are tested for a match. This proceeds recursively until a match is found on R_1; the R_t associated with that address is then used. For dense storage packing, the chain addresses are chosen from a list of empty positions, that is, R_m values which the existing vocabulary does not utilize.

It is possible to apply other transformations to R_1 to reduce it to the range from 1 to 2^M. A simple extraction of M bits may be practical. For any value of R_m, the chain should be assigned in order of decreasing frequency of word usage.

Storage is assigned by referring to its representation in three lists:

(1) Storage (prime) already assigned to a word
(2) Storage (secondary, or nonprime) already assigned to a word
(3) Free list, not yet assigned

A limiting number is experimentally chosen such that only this many words are allowed unlimited prime assignment. Starting with the most frequently used word, R_m is calculated and used as an address. If this address is found on the free list, it is removed and placed on the prime storage list. The address contents are assigned, using successively larger values of R_t. If this address is not found on the free list, a duplication in R_m has occurred. Such words are held aside for assignment after the limiting number is reached. These remaining words are then taken again in order of highest frequency of usage, and the remainder of the free list is used in sequence to fill the chaining addresses. Each word assigned must proceed through its chain. For example, take three words W_1, W_2 and W_3 for which the corresponding values of R_{1_1}, R_{1_2} and R_{1_3} all yield identical values of R_m.

W_1 is assigned	$(R_m) = R_{1_1}, R_{t_1}, FLA_1$
W_2 is assigned	$(FLA_1) = R_{1_2}, R_{t_2}, FLA_2$
W_3 is assigned	$(FLA_2) = R_{1_3}, R_{t_3}$, RETURN

(FLA means Free List Address)

Ends of all chains are assigned to the RETURN address. When a word must be added to the chain, it is lengthened by replacing RETURN by the next chaining address and putting RETURN in the new last word. If RETURN is ever reached in actual transmission, it indicates that this word is not yet in the dictionary. Automatic addition of this entry (in both sender and receiver) then occurs upon inspection of the free list.

Immediate Applications

Present-day computers operate at speeds too high for constant usage with communication lines, except under special circumstances. Until special devices are built for this express purpose, there are several ways of efficiently combining computers with existing communication lines.

(1) Message Center. Since the computer should be running nearly continuously to realize maximum savings from compression, one means of achieving economy is to create message centers in such cities as Paris, New York, London, etc., where the total volume of messages may be expected to approach capacity. Since the communication volume on lines is only about a third of present volume, capacity of such lines as Atlantic cables is tripled, without the need to lay new cables. Facsimile transmission may be interspersed with word messages to further justify the computer economy, since similar compression methods can achieve 4-to-1 reduction in this area [5].

The extreme flexibility of the computer allows a variety of modes of compression, as shown in Table 1. Some of these are suitable to the existing five-bit pattern of Teletype. Thus a computer equipped with paper tape input and output could take in continuous strings of normal Teletype messages, compress them, and output a continuous string in condensed form but still suitable for transmission on regular Teletype circuits and equipment. This tape then enters the receiving computer and is either printed on its equipment or converted to the expanded tape suitable for relaying to local Teletype receivers. Large networks could thus be two-stage, with the greater proportion of distance (and cost) being traveled in the compressed form. It is conceivable that an asymmetric condition could be used, with a central transmitter and several satellite receivers of lesser power. Radial transmission could be in compressed form, replies from the satellites in normal, uncompressed form. The compression algorithm must avoid using normal control codes as any part of the numbers. The same principles apply equally to other existing and proposed paper tape formats of six bits and more.

(2) Combined Message and Data Processing Center. Many computers are susceptible to external interruption of their regular operations and can intermingle several different jobs. This is known as multiprogramming. Thus a computer in regular operation can be interrupted upon demand to either encode a message for transmission or decode and print an incoming message. However, demand does not need to be heeded instantaneously. A minimum time lag would be that saved by the message compression. Depending on priority codes, lags of up to several minutes may be acceptable. This would allow for regular jobs to be interrupted at convenient points with minimum disruption.

(3) Combined Message and Language Translation Center. Several language translation schemes depend partially on corresponding dictionary lookup. In this method the receiving computer can look up the corresponding word in Russian or French just as easily as in English. Since this lookup time is such a small proportion of available real time, the rest of the translation process may be carried on simultaneously. This allows messages to be sent to multiple receivers in different languages through virtually instantaneous translation.

(4) Encoding for Secrecy. Secrecy comes virtually free with this code. Whereas ciphers depend upon letter

substitution and are normally broken on the basis of letter frequency in a language, it is quite another thing to try to determine relative frequencies for thousands of words rather than just 26 letters. Code is the more general term, being substitution of symbols for words.

Normally both sending and receiving computers are equipped with the same dictionary, or library. When the sender adds a new word to the dictionary, it must transmit this word to the receiver in both single character (so it will know what to print) and compressed form. Depending upon the mode of number assignment, this may cause a drastic restructuring of the entire encoding representation. In the example of the dictionary with words numbered in order, suppose a new word must be added just before "aardvark." Almost every word would have its representational number increased by one. This simple shift would be easy to detect, but the problem would be infinitely more difficult if many words were added at random places throughout the dictionary. Now imagine this ordering of numbers determined not by alphabetic ordering in the dictionary but by frequency of usage in the language.

Additions to the dictionary may be expected quite often. New users of the communications system may introduce new professional jargons. Personal or place names are used to identify many things, from army tactical positions to tropical storms. Mixed symbols such as part numbers for inventory will be used often. If these are popular or frequently used, it is more economical to add them to the compression dictionary than to send them by single characters. If not, the provision does exist to send single characters in groups of one, four, five, six or eight bits.

A particular business requiring secrecy could purchase its own special dictionary, scrambled in a unique way. Computers can store a multiplicity of these on tape files and both sending and receiving computers could select one upon the basis of a control code. Multi-programmed computers can merge several messages together for simultaneous transmission in a variety of patterns. A very simple example would be to interleave five messages so every fifth bit would belong to a specific message. Myriad varieties are possible and simple, but the unauthorized receiver would have to try every possible variety before he could make sense from any.

A valuable by-product of this method will be the ability (at last) to determine actual usage and frequency figures for both letters and words in languages. The compression program contains a counting mechanism for usage. This may be disconnected at option. This is useful to periodically rearrange the dictionary for efficiency, when operating in a standard non-secret mode. Previous counts, on sample texts of from 100,000 to 300,000 words, did not count punctuation symbols for frequency of occurrence. In this method, it is more economical to use these symbols as being identical to words.

TABLE 4.
EXAMPLE OF POSSIBLE ASSIGNMENTS (Type D)

Octal	Symbol	Frequency*	Octal	Symbol	Frequency*
00	(Open for contingency)		40	ARE	1200
01	Enter binary mode		41	ON	1200
02	Enter 4-bit (decimal) mode**		42	OR	1100
03	Enter 6-bit mode**		43	HER	1100
04	Enter 8-bit mode**		44	HAD	1100
05	Blank		45	AT	1100
06	,		46	FROM	1000
07	THE	15500	47	THIS	1000
10	.		50	MY	1000
11	OF	9800	51	THEY	1000
12	AND	7600	52	ALL	900
13	TO	5700	53	THEIR	800
14	A	5100	54	AN	800
15	IN	4300	55	SHE	800
16	THAT	3000	56	HAS	800
17	IS	2500	57	WERE	800
20	I	2300	60	ME	700
21	IT	2300	61	BEEN	700
22	;		62	HIM	700
23	FOR	1900	63	ONE	700
24	AS	1900	64	SO	700
25	WITH	1900	65	IF	700
26	WAS	1800	66	WILL	700
27	HIS	1700	67	THERE	700
30	HE	1700	70	WHO	700
31	BE	1500	71	NO	700
32	NOT	1500	72	WE	600
33	BY	1400	73	WHEN	600
34	BUT	1400	74	WHAT	600
35	HAVE	1300	75	YOUR	600
36	YOU	1300	76	MORE	600
37	WHICH	1300	77	(Open for contingency)	

* 242,000 word sample from *Cryptanalysis*, H. F. Gaines, Dover, 1956.

** 4-, 6- and 8-bit modes have identical characters to IBM LOGICODE proposal. Those modes are provided to allow single character formation. The 4-bit set is provided with a special blank following 0 — 9 . — + . Return to Normal Mode is effected in the various sets by encountering the character

01111100 in the 8-bit set
111110 in the 6-bit set (If 6-bit return character is out-of-phase with the end of the byte, hold up 2 bits)
1110 in the 4-bit set

Amortization of Computer Costs

A rough program planned for an IBM 7090 takes an average of 250 microseconds per word for total conversion at both transmitting and receiving ends. This rate of about 4000 words/second is more than adequate to keep up with foreseeable transmission times, even microwave, on a real time basis. At a nominal cost of $800 per hour, the per-word cost would be

$$\frac{80000}{3600 \times 4000} = .0056 \text{ cents per word.}$$

Cost estimates for land line transmission are 62 cents for an average message consisting of 48 words of 5 characters and blank, requiring an overall elapsed time of one minute. These 288 characters actually require only 14.4 seconds on the line at 20 characters per second transmission rate. To realize the case most unfavorable to computer terminals, assume interleaving of four messages. The cost would be

$$\frac{62}{4 \times 48} = .323 \text{ cents per word.}$$

This shows conversion time to be negligible by almost two orders of magnitude to the transmission costs saved. Net costs with computers on each end should approximate

$$(35\% \text{ of } .323) + .0056 = .1186 \text{ cents per word.}$$

Thus the overall cost may be expected (with a fully utilized computer) to be from 37% to 40% of present costs. The most profit comes from transoceanic routes rather than land lines. Some typical costs* are:

Radio/Cable	New York to London	21 cents per word
	New York to Paris	25 cents per word
	New York to Moscow	25 cents per word
	New York to S. America	31 cents per word
	New York to Japan	34 cents per word
Western Union-Telex	New York to London/Paris	$9 for first three minutes, $3 each additional; 66 words/minute at 50 bauds

*(night rates are half of day rates)

Straight text costs a standard amount per word based upon average word length. Coded text is charged modulo 5 letters. A group of 5 letters or less counts as one word; a group of more than 5 is counted by components of 5 and fewer letters. Thus a 12-letter group would be charged as three words.

REFERENCES

1. Draft Standard 7233: 1–4: 5/60, Electronic Industries Association, Committee TR 24.4.
2. Shannon, C. E., *Bell System. Tech. J. 30* (1951), 50–58.
3. Brillouin, L., *Science and Information Theory,* pp. 24–58 (Academic Press, New York, 1956).
4. Dewey, G. C., *Relativ Frequency of English Speech Sounds* (Harvard University Press, 1923).
5. Wyle, Erb and Banow, Reduced-time Facsimile Transmission by Digital Coding, Preprint at National Symposium on Global Communications, August 1960. (Ford Instrument Co., Long Island City 1, N.Y.)

Survey of Coded Character Representation

R. W. Bemer, *IBM Corp., White Plains, N. Y.*

Technical Committee 97 of the International Standards Organization (ISO) is concerned with standards in data processing. The American Standards Association holds the secretariat of this committee. Sectional Committee X3 of A.S.A. is responsible for national data processing standards in the U. S.

One of the most important areas of standardization (and one of the most pressing) is that of the logical representation of the character sets. These representations may be by punched holes in paper tape, pulses on a communications line, bits stored in memory, marks on paper, etc. The existing standards work in this area has been done by MEE 149 (now DPE) of the British Standards Institution and by TR 24.4 (now TR 27.6) of the Electronic Industries Association. This particular problem is now under the cognizance of A.S.A. Sub-committee X3.2 on character sets and data format.

The chart (pp. 640–1) is presented as staff work for the deliberations of X3.2. It is the most complete information we have been able to assemble to date, but obviously there may be errors and omissions. The primary aims in publishing this chart are:

(1) To indicate to the information processing industry why standardization is vital in this area.

(2) To request further information from the various experts who possess it.

The chart is presented in 64-character modules (sufficient for a 6-bit set) and the positions are given designations in the octal number system from 00 to 77. For the benefit of anyone not familiar with this notation, a conversion table is given:

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

"1" represents an ON or a punched hole condition. "0" represents OFF or an unpunched condition. Obviously, the conversion between physical and logical representation may be made in two ways (for example, depending upon which edge of the paper tape is on your right hand).

Five-track paper tape is shown as though one of either the letter or figure shifts had a sixth track. In most cases this is theoretical. The letter shift-figure shift relationship is indicated on the chart together with the tape orientation. If a "3" appears in the column marked units, then the "3" hole side is octal 01. If a "2" appears in the units column the reverse is true. In most British tapes the letter shift is shown in the high position because the collating sequence adopted by the British for six-track codes puts the digits low to the alphabet. The converse is mostly true in the U. S.

Note: The 704 and 705 codes, for example, apparently violate this statement since the digits have lower octal representations than the alphabet. *However*, ordering of files is controlled by a collating sequence in which the digits are higher than the alphabet. This is accomplished either by a comparison matrix in hardware or by programmed replacement of the keys in the records to be ordered.

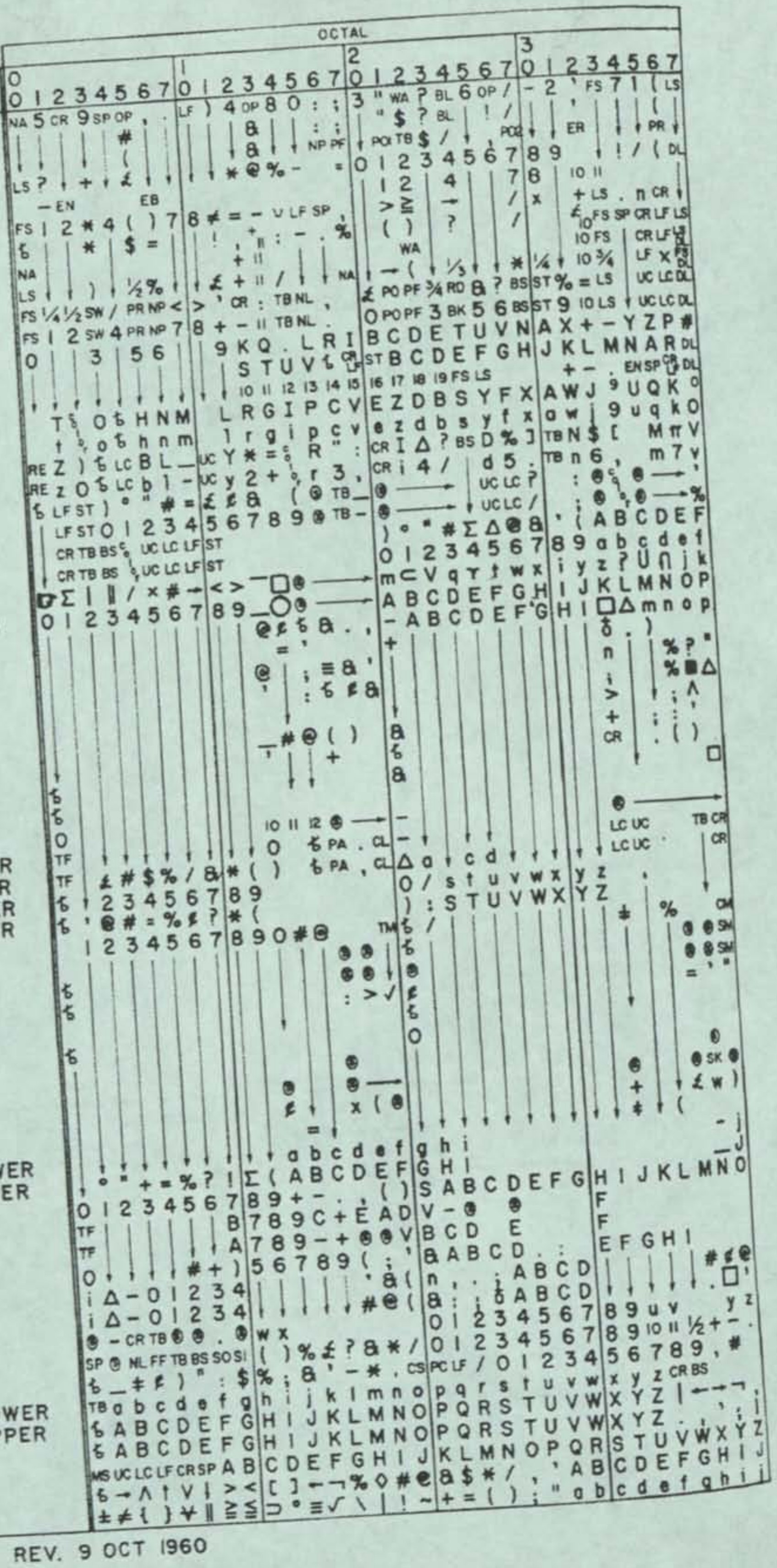
Some seven element codes are shown on two lines. In most cases these are accomplished by an upper and lower case shift on the input key board. These cases are indicated. The 7030 actually uses the seven (and eight) bit set internally.

The elements of the code sets may possess either informational or control characteristics. In my personal opinion they should not possess both. Informational characters are shown by their single graphics. Control and functional characters are coded with two-letter mnemonics according to the following table, except for blank (which is shown as a lower case *b* with a slash through the stem) and special (Ⓢ).

BK	black
BL	bell
BS	backspace
CL	clear
CM	card mark
CO	compute
CR	carriage return
CS	carriage shift
DL	delete (erase)
EB	end block
ED	end data
EF	end file
EI	end information
EN	end number
ER	error
ES	escape
FF	form feed (paper throw)
FS	figure shift
ID	idle
LC	lower case
LF	line feed
LS	letter shift
MS	master space
NA	no action
NL	new line (CR + LF)
NP	non-print
OP	optional
PA	put away
PC	page change
PF	punch-off
PO	punch-on
PR	print restore
RD	red
RE	read
SI	shift in
SK	skip
SM	segment mark
SO	shift out
SP	space
ST	stop
TB	tabulate
TF	tape feed
TM	tape mark
UC	upper case
WA	who are you?

Survey of Coded Character Representation

MACHINE	CASE
C.C.I.T.T.	
TELETYPE	
FLEXOWRITER	
METRO-VICK 950	
LEO	
PEGASUS - MERCURY	
ELLIOTT 405	
B.S.I. DRAFT - MAY '59	
E.M.I. 1100, 2400	
PEGASUS FLEXOWRITER	UPPER
" "	LOWER
STANTEC ZEBRA	
E.M.I. M/C TOOL	
ENGLISH ELEC. DEUCE	
1103A TYPEWRITER	UPPER
" "	LOWER
LGP-30 FLEXOWRITER	UPPER
" "	LOWER
RPC - 9000	UPPER
"	LOWER
RPC - 4000	UPPER
"	LOWER
LINCOLNWRITER	UPPER
"	LOWER
NCR - 304	
704, 709, 7090	
PHILCO 2000	
M-H 800	
BENDIX I/O TYPER	
RCA 301	
M-H 1000	
ALWAC III (CARD)	
BENDIX G15 (CA-2)	
PERSEUS	
NCR PAPER TAPE TYPER	LOWER
" " " "	UPPER
PDP-1	LOWER
"	UPPER
705, 7080, CDC 1604	
7070 MAG. TAPE	
1401	
1410	
650 MAG. TAPE	
305 RAMAC	
BURROUGHS 220 P-TAPE	
IBM 046 "	
NCR "	
IBM 1620 "	
ALWAC III FLEXOWRITER	LOWER
" "	UPPER
HIDAC 101 PAPER TAPE	
KIMBALL PUNCH TAG	
DENNISON " "	
USS-80	
UNIVAC I, II	
UNIVAC III	
BENDIX G15	
FERRANTI PROPOSAL (B.S.I.)	
RCA 501, BIZMAC	
BENDIX G 20 TYPEWRITER	LOWER
" " "	UPPER
GAMMA 60	
FIELDATA, MOBIDIC	
IBM 7030	
" "	

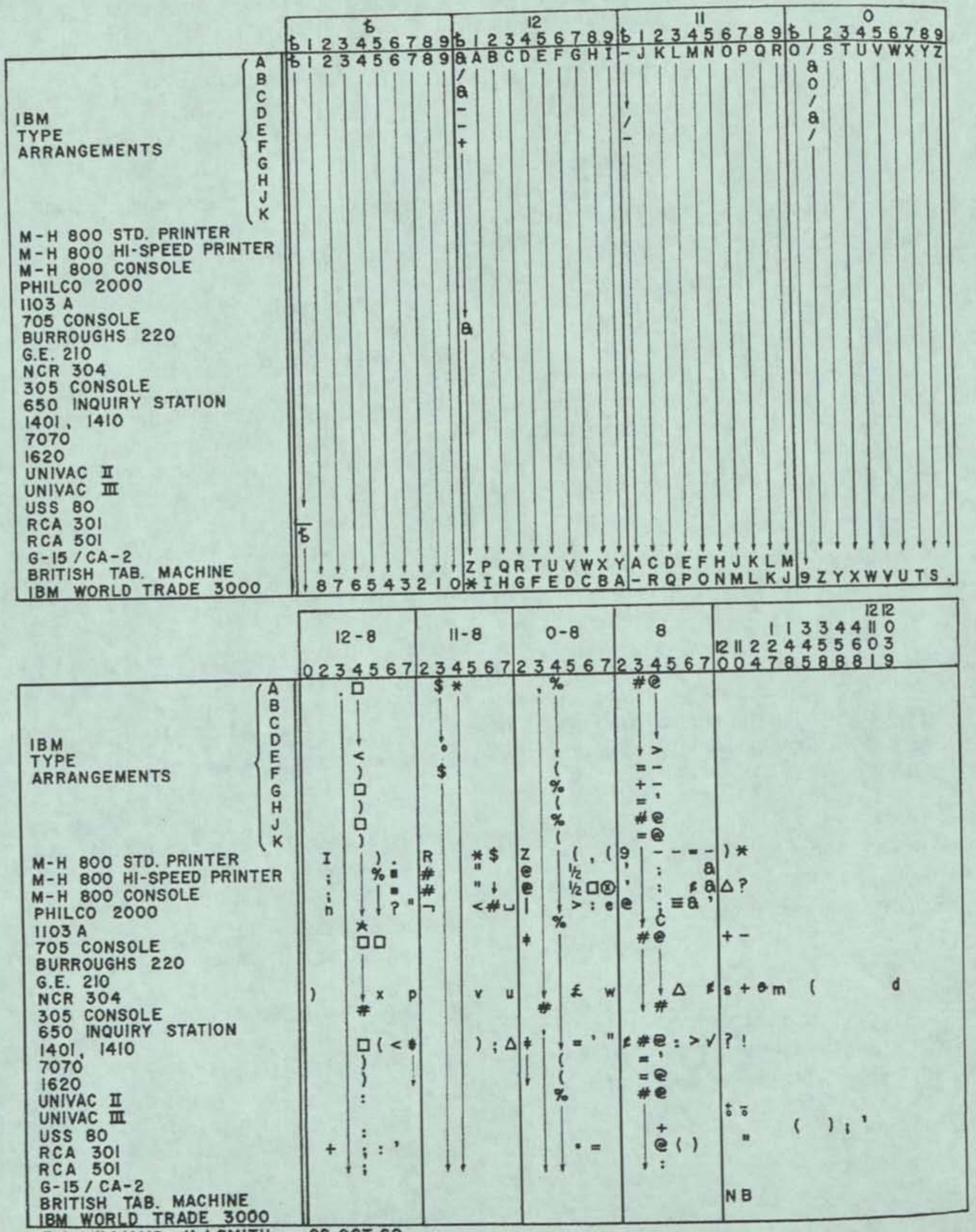

OCTAL
0 1 2 3
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

R.W. BEMER, F.A. WILLIAMS

REV. 9 OCT 1960

OCTAL																																SHIFT	UNITS	MACHINE
4								5								6								7										
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7			
MS	T	CR	O	SP	H	N	M	LF	L	R	G	I	P	C	V	E	Z	D	B	S	Y	F	X	A	W	J	FS	U	Q	K	LS	FIG – LTR	3	C.C.I.T.T.
ST	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	LS	↓	3	TELETYPE
LS	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	3	FLEXOWRITER
																↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	DL	↓	3	METRO-VICK 950
FS	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	LS	.	?	£	DL	FIG ONLY	3	LEO
NA	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	FS	SP	CR	LF	LS DL	FIG – LTR	2	PEGASUS - MERCURY
NA	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	FS	↓	CR	LF	LS DL	↓	3	ELLIOTT 405
LS	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	&	↓	LF	'	FS DL	↓	3	B.S.I. DRAFT - MAY '59
FS	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	LS	↓	UC	LC	DL	↓		E.M.I. 1100, 2400
FS	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	LS	↓	UC	LC	DL	↓	2	PEGASUS FLEXOWRITER
																																↓	"	"
																																	2	STANTEC ZEBRA
																																	2	E.M.I. M/C TOOL
SP	A	B	C	D	E	F	G	H	I	.	J	K	L	M	N	O	P	Q	R	FS	LS	S	T	U	V	W	X	Y	Z	CR LF	DL	FIG – LTR	3	ENGLISH ELEC. DEUCE
		)	ST	×	CR	(	UC	_	TB	'	/			−	LC	8	BS	5		4		6		3		7		2			↓			1103A TYPEWRITER
		.	ST	=	CR	,	UC	|	TB	(	+			−	LC	8	BS	5		4		6		3		7		2			↓			" "
ST	P	Σ	O		E	(	X		U	F			T	G			H	J			C	K			A	Q			S	W	↓			LGP-30 FLEXOWRITER
ST	p	8	o		e	9	x		u	f			t	g			h	j			c	k			a	q			s	w	↓			" "
.														|	+	÷															↓			RPC-9000
.	A	B	C	D	E	F	G	H	I	J	K	L	M	*	$	@	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	'	↓			"
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	$	:	;	%			?	_	.	SP	÷	↓			RPC-4000
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	,	=	[	]			+	−	.	SP	/	↓			"
α	Δ	ρ	ε	κ	⊃	β	∧	λ	~	{	}	≡	=	'	*	CR	TB	BS	⊗	→	→	→	→	→	→	→	→	LC	UC	ST	↓			LINCOLNWRITER
Q	R	S	T	U	V	W	X	Y	Z	(	)	+	−	,	.	CR	TB	BS	●	→	→	→	→	→	→	→	→	LC	UC	ST	↓			"
+	J	K	L	M	N	O	P	Q	R	%	£	$	(	)	/	*	#	S	T	U	V	W	X	Y	Z	d	s	u	v	w	x			NCR-304
−	↓	↓	↓	↓	↓	↓	↓	↓	↓	ŏ	$	*				ƀ	/	↓	↓	↓	↓	↓	↓	↓	↓	‡	,	(						704, 709, 7090
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	¬	↓	↓	<	#	⊔	Δ	↓	↓	↓	↓	↓	↓	↓	↓	↓	|	↓	↓	>	:	e			PHILCO 2000
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	#	↓	↓	=		?	CR	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	↓	↓	½	□	⊗			M-H 800
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	CR	↓	↓		TB		°	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓	SP		DL			BENDIX I/O TYPER
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	EI	↓	↓		ED	EF	"	↓	↓	↓	↓	↓	↓	↓	↓	↓	EB	↓	%	•	=				RCA 301
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	−	↓	↓		¢		&	↓	↓	↓	↓	↓	↓	↓	↓	↓	:	↓	↓	½					M-H 1000
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓					↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓						ALWAC III (CARD)
	↓	↓	↓	↓	↓	↓	↓	↓	↓									↓	↓	↓	↓	↓	↓	↓	↓									BENDIX G 15 (CA-2)
.	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	→	→	→	→	→	ƀ	S	T	U	V	W	X	Y	Z	@	→	→	→	→	→	→			PERSEUS
@	↓	↓	↓	↓	↓	↓	↓	↓	↓					CO		¢		S	T	U	V	W	X	Y	Z						DL		3	NCR PAPER TAPE TYPER
+	↓	↓	↓	↓	↓	↓	↓	↓	↓					ST		□		s	T	u	v	w	x	Y	Z						DL		"	" " " "
−	j	k	l	m	n	o	p	q	r		$					&	a	b	c	d	e	f	g	h	i	LC	.	UC			CR			PDP-1
"	J	K	L	M	N	O	P	Q	R		−					;	A	B	C	D	E	F	G	H	I	LC	↓	UC			CR			"
−	↓	↓	↓	↓	↓	↓	↓	↓	↓	ŏ	$	*				&	↓	↓	↓	↓	↓	↓	↓	↓	↓	+0	↓	⌑			‡			705, 7080, CDC 1604
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	@	Δ	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	@	↓			7070 MAG. TAPE
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	@	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	@	↓			1401
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	!	↓	↓	)	;	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	?	↓	↓	(	>	↓			1410
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓				↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓						650 MAG. TAPE
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓				↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓						305 RAMAC
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓	@	@		↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	↓	↓			@			BURROUGHS 220 P-TAPE
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	↓	↓	@	CR	ER	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	@	↓	↓	@		DL			IBM 046 "
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	Δ	↓	↓	@	→	→	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	PA	↓	↓	d	CO	DL			NCR "
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	↓				+	↓	↓	↓	↓	↓	↓	↓	↓	↓		↓	)						IBM 1620 "
k	l	m	n	o	p	q	r	LC	UC	%	DL	TB	CR	BS		0	/	s	t	u	v	w	x	y	z	$	|	ST		,	.			ALWAC III FLEXOWRITER
K	L	M	N	O	P	Q	R	LC	UC	%	DL	TB	CR	BS		)	'	S	T	U	V	W	X	Y	Z	Δ	*	ST		;	:			" "
X	P	Q	R	T	Y	CR	Z	LS	U	V	W																							HIDAC 101 PAPER TAPE
																																		KIMBALL PUNCH TAG
																																		DENNISON " "
−	J	K	L	M	$	*		N	O	P	Q	R				ƀ	/	S	T	U	,	%		V	W	X	Y	Z						USS-80
†	"	]	)	J	K	L	M	↓	↓	↓	↓	↓	$	*	?	Σ	β	:	+	/	S	T	U	↓	↓	↓	↓	↓	%	=	δ			UNIVAC I, II
†	"	)	ŏ	J	K	L	M	↓	↓	↓	↓	↓	$	*		Σ		=	+	/	S	T	U	↓	↓	↓	↓	↓	,	%				UNIVAC III
																⊙																		BENDIX G 15
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z				ES	DL			FERRANTI PROPOSAL (B.S.I.)
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	EF	ED	•	>	<	DL			RCA 501, BIZMAC
0	1	2	3	4	5	6	7	8	9	10	.	+	−	*	/	=	∨	≠	∧	<	$	>	:	(	[	]	)	↓	↑	:	'			BENDIX G 20 TYPEWRITER
																																		" " "
0	1	2	3	4	5	6	7	8	9	+	−	(	)	×	/	=	≠	>	<	Δ	&	π	Σ	%	£	$	*							GAMMA 60
)	−	+	<	=	>	_	$	*	(	"	:	?	!	,	⊕	0	1	2	3	4	5	6	7	8	9	'	;	/	.	□	ID			FIELDATA, MOBIDIC
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	0	1	2	3	4	5	6	7	8	9	.	−							IBM 7030
k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	0	1	2	3	4	5	6	7	8	9	:	?							" "

Survey of Punched Card Codes



b̸ | 12 | 11 | 0

b̸ 1 2 3 4 5 6 7 8 9 | b̸ 1 2 3 4 5 6 7 8 9 | b̸ 1 2 3 4 5 6 7 8 9 | b̸ 1 2 3 4 5 6 7 8 9

IBM TYPE ARRANGEMENTS A: b̸ 1 2 3 4 5 6 7 8 9 | & A B C D E F G H I | - J K L M N O P Q R | 0 / S T U V W X Y Z

BRITISH TAB. MACHINE: Z P Q R T U V W X Y | A C D E F H J K L M

IBM WORLD TRADE 3000: 8 7 6 5 4 3 2 1 0 | * I H G F E D C B A | - R Q P O N M L K J | 9 Z Y X W V U T S .

12-8 | 11-8 | 0-8 | 8

0 2 3 4 5 6 7 | 2 3 4 5 6 7 | 2 3 4 5 6 7 | 2 3 4 5 6 7

A
B
C
D
E
F
G
H
J
K
M-H 800 STD. PRINTER
M-H 800 HI-SPEED PRINTER
M-H 800 CONSOLE
PHILCO 2000
1103 A
705 CONSOLE
BURROUGHS 220
G.E. 210
NCR 304
305 CONSOLE
650 INQUIRY STATION
1401, 1410
7070
1620
UNIVAC II
UNIVAC III
USS 80
RCA 301
RCA 501
G-15 / CA-2
BRITISH TAB. MACHINE
IBM WORLD TRADE 3000

F.A. WILLIAMS, H.J.SMITH 28 OCT 60

Reprinted from The Computer Bulletin, Vol. 4, No. 4

R Bemer

SURVEY OF MODERN PROGRAMMING TECHNIQUES

by R. W. Bemer

This paper formed the basis of a talk given to the Society following the Annual General Meeting on 29 September 1960.

Introduction

In the business section of the *New York Times* there often appears an advertisement (not of my own company) for "Research Programmers to work in Macro-Assembly language development, Heuristic Programming and Artificial Intelligence studies, Symbol Manipulation and other advanced computer areas." You may have seen it. Even the lowly Machine Programmers are requested to "write programs for a variety of large-scale digital computers in the areas of Scientific Information Processing, Natural Language Processing and Information Retrieval Systems." At first this may sound like somebody has been reading Mr. Potter's books and this is merely one-upmanship in the programming area, but I assure you this is not so. Programming has indeed moved to glamorous heights.

Until about four years ago, programming was a more homogenised profession. This was to be expected in a relatively new field. However, so were the developments outlined in this advertisement. At present we have a large number of programmers in the world, certainly over 30,000, and the techniques used range from the ones described down to the most archaic. It is extremely unfortunate that the archaic end is the large end of the iceberg—the part under water. This is occasioned by the sheer rise in production programming, particularly in (but not restricted to) business and scientific applications. The production of generalised systems such as the FORTRANS, Flowmatics, various assembly programs, and rather complete systems like SOS for the 709 is a very big business. I hope that the end-purpose of this talk (and others like it, with published articles on the topic) will be to raise this vast body of programmers from the doldrums of outmoded techniques. I realised in 1950, after my first year with electronic computers, that the leverage factor between a good and bad programmer, or a good and bad technique, can easily be as high as ten or twenty-to-one. In a tricycle factory one is likely to become vice-president for increasing the output 10% at the same manufacturing cost. In programming, a 10% betterment of efficiency—that is in construction, not running efficiency—is likely to go unnoticed.

I shall try, in this talk, to give a summary of new and improved techniques in the programming field. Surveys should gather information in one place to enable proper perspective for review and weighting of importance. This survey will be restricted to generalised techniques and tools. Applications will not be covered, otherwise you might not get home until breakfast tomorrow morning. Despite much necessary overlapping, I am going to divide this talk into six parts as follows:

1. The elements of languages
2. Machine-dependent languages
3. Machine-independent languages
4. Analysis languages
5. Processor techniques
6. Operating systems.

Elements of Languages

When we instruct the computer to do work it is analogous to instructing another human being. In both cases we use languages. In early attempts the languages were at a very crude level and very awkward to use. Much of the recent pressure has been to use English as the language medium and instruct the machine almost indistinguishably from the instruction of another human. I have severe doubts as to whether we can or should go in this direction alone. One thing is very sure—the economic need to more efficiently communicate with machines has provided great pressure to re-examine the meaning and structure of language. Millions of us use the English language quite correctly, or at least as correctly as most, by having learned it through example and unconscious statistical selection. It may be possible that some day we will also teach machines in this way, but with present machine construction this is likely to be very expensive. Most of our present approach is devoted to teaching languages by a rigorous exposition of their form and structure. [illegible], ETC.

There are many types of languages, and I don't mean Russian, French, or Pakistanian. There are the linear languages such as we have in writing or speech. There are the two-dimensional languages of tables and lists. There are symbol languages, such as flowcharts, and these by implication may be in many dimensions. There are pictorial languages. All of these have been used to communicate with computers.

ESPERANTO/INTERLINGUA/MONEY MOTIVE—DOUBL SALARY

The formation of language symbols is most interesting. Most all symbols have recursive, or combining properties. For example, the Chinese symbol for "riot" is formed of two identical symbols for "woman," with a broken line across to indicate a roof. Thus, to any knowledgeable man, two women under one roof indicates a riot.

Alphabets, however, are far more efficient, and have beautifully recursive properties. Except for a few vowels, none of the single characters are meaningful words in English, disregarding their use as single-character symbols for mathematics and the like. Theoretically—and I say theoretically only because George Bernard Shaw would otherwise arise from his grave—the characters in groups of one, two and three, etc., all have corresponding verbal sounds. These verbal sounds are then the analogue representations of symbols. However, I am afraid it would be very difficult to speak English to an analogue computer. The cost of storing symbol patterns for discrimination would be horrendous.

VOCODER

STUTTER

The digital computer is more fortunate because it can use binary bit representations for the elements of language. In common usage, bit representations are assigned to the letters of the alphabet, decimal digits and other useful characters. Using information theory, Shannon and others (myself among them) have used bit representations for entire words or phrases. However, to add new words or names the facility must always exist to represent the single characters by unique bit combinations. An analogy may be found in the representation of numbers by both coded decimal and binary notation. The binary notation corresponds to the word, since the entire symbol (that is, quantity) is represented by a single number, even though that has recursive forming elements of 0 and 1. The decimal number, or of any other base for that matter, is formed recursively by adjoining number symbols instead of letters. Let us not overlook the combinations of both numbers, letters, and special symbols, useful for part numbers and automobile licence numbers.

If I may speak categorically, the input to and the output from a computer is primarily a bit stream. Whether or not this bit stream is broken up into bytes, of which some bits are delayed in time so that a group or byte of bits enters the computer in parallel, is of no consequence. The size of such bytes, whether they be defined as a single bit byte, the 5-bit Teletype or Baudot code, the 6-bit byte of many alphabetic computers, or the 8-bit byte of certain new computers and numerically controlled machine tools is of consequence only to the convenience of the designer and the efficiency of his product. I recommend for your study the paper by Howard Smith, Jr., of my group, which appeared in the August 1960 issue of the COMMUNICATIONS of the *ACM*, entitled "A Short Study of Notation Efficiency."

Some of you may know that I have been crusading for some time in the interest of larger character sets. This has met with some success and you may note the IBM 7030—the production version of the STRETCH computer—accommodates a 256 character set, 8 bits per character. The attached printer will print 120 characters, including the upper and lower case alphabets, and all the other characters of the reference language of ALGOL. The input/output typewriter for the Bendix G20 also handles 8-bit ALGOL characters. This is of great interest to the programmer because he may now identify the unequivocal meaning of each character in a string without resorting to long programs that make many decisions on contextual relationships.

54

To say there has been variety in the methods of assigning bit combinations to characters is putting it very mildly. We have catalogued over 50 different selections. This Babel has probably been the prime factor in instigating an international standardisation effort in data processing. The British Standards Institution has been active here for years. In the United States, consideration of these problems has been left until recently to the various professional and trade organisations. However, the X3 Sectional Committee for data processing has been formed in the American Standards Association to straighten out this matter and many others at both the national and international level. Quite naturally, there is excellent co-operation between the British Standards Institution and the American Standards Association in these matters.

CAREFUL — NEG. FACTIONS

In actual practice, the bit representations do not need to be identical for interchange of information. The basic need is for a uniform collating or ordering sequence of characters. There is nothing more vital to the interchange of data and programs between computing machines than an identical collating sequence. There are certain natural collating sequences (Z is higher than A, 9 is higher than 0). I know of no reason why alphabet should be higher than numbers or numbers higher than alphabet other than historical precedent. Most of the millions of files produced by data processing machines in the United States are ordered with the numbers higher than the alphabet. Blank is a character in its own right and must be low. Collating sequence is an important factor in machine cost. Unless the ascending binary sequence of characters in machine representation is the same as the collating sequence, additional and expensive hardware will be needed to compare the keys of items to be marshalled or ordered.

Another basic element in interchangeability is data format. In order to be operated upon, data must be precisely defined. This definition may be by means of the instruction sequence itself, by other stored data, as in the control word technique, or by self-definition. For the latter purpose, I prefer a numeric subset of the 4-bit characters that contain the decimal digit 0 through 9, decimal point, minus, plus, comma, blank, and perhaps a monetary sign, dollars or pounds. In most present arithmetic operations the position of the decimal or binary point is accommodated by either floating point arithmetic or by aligning the implicit decimal point within the instruction sequence. This alignment could possibly be done automatically with a coincidence-matrix detector if the decimal point is an explicit and separate character contained in the data. Some scaling might still be necessary, of course. Another type of instruction/data interaction is that where the data itself signals that it is of a special class which may or may not alter the instruction sequence. An example of this would be the terminator bits in the 7030, which indicate the beginning and ending elements of a vector stored in memory. TRAPPING, STREAMS

Although another element of language structure is the syntax, I will take that up under some other groupings.

Machine-Dependent Languages

As long as we have computing machinery there will be a machine language for a particular computer to understand. I will not guarantee that the form will stay that way it is today, because already

1. There are fixed word length and variable word length machines.
2. There are machines that operate on words, machines that operate on characters, and machines that operate on bit streams.
3. There are machines of one command and of more than one command in a single instruction, with one, two, three, four, and perhaps more addresses in a single instruction.
4. There are machines with 20 instructions in the machine language repertoire and machines with over 500 different types of commands available.
5. There are micro-instruction languages with which the programmer can get at each primitive required in fetching the necessary data to perform the operation; there are machines which have macro-instructions built into the hardware when a high frequency of usage indicates enough gain by paralleling the elements of execution. Examples of the latter are floating point, operations involving index registers, operations of indirect addressing and special table instructions such as the convert instruction in the IBM 709.

NATURE
≠
GROWTH
VS
IMPOSED

SHAVER
EMPLOYER
RULES

We may yet see machine languages identical to ALGOL or some other presently machine-independent language.

Expert programmers are well aware of the uncertainty in machine languages of the future. One certainty is that at the present time the engineers are far outstripping the ability of the programmer to use the machine, and there is a saturation point beyond which no amount of programmers can possibly speed up the writing of a program. Certainly more than two or three hundred programmers working together constitute a point of diminishing returns. We have no recourse as programmers but to go to the machine designer and say "help." I am pleased to note that the Atlas machine has taken many steps forward in this direction. The convenience of numerical symbolic addressing is one of the most important features that will reduce translation time and programming effort.

Given a particular machine-dependent language, there are many interesting tricks and techniques which a programmer may use, sub-programs for counting the number of 1's in a binary number, for instance, or tricky mathematical sub-routines. Although not exactly applications, such techniques are nevertheless also excluded from this treatment. Let us leave machine-dependent languages with only the reservation that eventually we have to convert the information we give the machine into this form. NO POSSIBILITY OF STDS.

Machine-Independent Languages

Machine-independent languages may be divided into two groups—and I do not mean scientific versus commercial languages. This is an entirely different partitioning. Machine-independent languages are either *procedure-oriented* or *problem-oriented.*

There is a great deal of confusion existing between these two terms. It is unnecessary confusion because the distinction is simple. The procedure-oriented languages are available for one to describe *how* the process is to be carried out. With the problem-oriented language one needs only state the problem. Heuristic programming is of course only the upper stratosphere of problem-oriented languages. There are many of these in existence today, of a simpler nature. As an example, take what is miscalled (in the United States) a sort-generator. What they really mean is ordering, or, to use the British term, marshalling. The input to such a generator would be items such as internal memory size, number of tape units, suspected bias in the ordering, record size and layout for the items to be ordered, preference for ordering method, grouping or blocking information, and many other items of information or advice. Inherent in the sorting generator is the pseudo-intelligence about the problem which will, from the intersection of this information and certain basic skeletal routines, construct an efficient operating program. The programmer may have called for a distribution or a sifting sort. He did not tell the machine how to accomplish a distribution or sifting sort. Had he actually written the program for a distribution sort he would probably have done so in procedure-oriented language. ALGORITHM ≠ RECIPE

Many of you have undoubtedly noted the metalinguistic formulae in which ALGOL 60 is described. This is due to John Backus, previously known as the developer of the FORTRAN program and language. I trust I may be excused for considering this a tremendously more important development than FORTRAN. Algebraic languages did exist before FORTRAN—Rutishauser's and that of Laning and Zierler at *MIT*. I believe Brooker's work was also simultaneous with FORTRAN.

This meta-language seems to me a remarkably rigorous means of describing a linear or string language. One would assume that the process should be recursive. That is, there should be a meta-meta-language with which to describe the meta-language, and so on in depth. I have always been convinced that such rigorous formation rules tend to simplify the translation process, just as in working at the aircraft factories during the war I found that the lower degree of the profile curve, the

BARBER SH
CAVER
SET/MEMBER

BINARY STAR, BUT "[illegible]" SAYS ROCKET, WORK
CT = COM'L TRANSPORT
CODEL =
FM = BIRD FEEDER

better the air liked it. I was particularly pleased to hear from Peter Ingerman, University of Pennsylvania, at an ALGOL discussion during this summer's ACM meeting, that they had some difficulty implementing non-recursive procedures in ALGOL. When they redefined the procedures to be recursive everything was much simpler. In other words, a sounder and more generalised structure forces the programmer to do things the right way.

TEACH MACHINES— TASCHENBUCH

You are by now all familiar with the trend in scientific machine-independent languages through FORTRAN, UNICODE, Math-matic, Auto Code and such to the present state with ALGOL 60. Although by no means complete (in fact, I consider it still quite experimental) ALGOL is a far superior language to any of its predecessors. I know of four related processors for ALGOL in Germany; in the United States, processors have been written at least for ALGOL-like languages for the Burroughs 220, the CDC 1604, and its prototype Countess. An ALGOL processor exists for the 709/7090, and ALGOL processors are being constructed for many other machines. I have enough faith in the eventual future of ALGOL to have caused a program to be constructed which converts from FORTRAN source language into a rather stupid ALGOL. I have been asked many times why we did not make it translate from ALGOL to FORTRAN so that the existing processors could be utilised. The answer has always been that we wish to obsolete FORTRAN and scrap it, not perpetuate it. Its purpose has been served.

MANY = CHOICES, TRANSL AFTER MAKING

A similar revolution is now taking place in the area of business languages. Under the sponsorship of the US Department of Defence there has been formed the Conference on Data Systems Languages (CODASYL). Although this conference has other long-range aims, its initial and most urgent purpose was to synthesise, from the existing business languages such as Flowmatic, Aimaco, and Commercial Translator, a somewhat universal language in the spirit of ALGOL.

This language, COBOL, is nearly complete in its definition. Its construction was beset with many more difficulties even than ALGOL. For one thing it had to handle almost all the features and classes of problems that ALGOL does in addition to many others. Let there be no mistake about it—business and commercial problems are vastly more difficult of solution than are scientific problems, at least in their translation to machine operation. The scientists and mathematicians, in constructing ALGOL, drew upon a workable language of mathematics that has been in existence for hundreds of years. Their new contribution was the reduction of the verbiage that the mathematician normally finds between the formulae to algorithmic form in a more concise notation. On the contrary, business practice has differed wildly.

The constructors of the COBOL language were beset by many new problems and I fear that in their initial attempt they ignored the rigour and syntactic beauty that a definition by meta-language would have gained them. There has been a general resistance on the part of *IBM* and myself to the willy-nilly adoption of COBOL in its original form. We knew what was wrong with it and tried to say so in the manner of elder statesmen. I am pleased to say that nearly all these basic flaws have now been removed. *IBM* is committed to produce COBOL processors for many of its computers on the assumption that the official form of the language will be revised no oftener than once a year. Practically all major producers of computing equipment in the United States are committed to COBOL processors for their machines.

One might now ask if ALGOL and COBOL are the end. I must say no, for part of the work the American Standards Association set up under its X3 Committee is a project for common programming languages. I suspect there will be those who walk into the X3–4 Sub-Committee and expect to find ALGOL adopted as a standard. I expect the same may be true for the COBOL proponents. Having played the scientific against the commercial and vice versa, Saul Gorn and I have reason to believe that this is the very lever needed to force a fusion into a single language for both scientific and commercial work.

GA UNIV. LANG.

If machine-independent languages are to be standard, they must be standardised according to a set of rules of graduated stringency. Adoption of a particular existing language as a standard would be fallacious. For one thing, a standard requirement should be that the language be expressible in the meta-language of Backus or some other development of this nature. For another, all languages should be clearly partitioned. The commercial languages are now in three parts, reminiscent of Gaul (!); namely, procedure, data description, and environment. ALGOL does not have separate data description because it operates only upon floating-point variables or fixed-point variables with rigid rounding and truncation principles not suitable to business. ALGOL does not have an environment section, and it could certainly use it.

PROGRAMMR EASE, ELIMINATE REPETITION

I further suspect within a period of two years a fourth section will be broken out of the language, a section exclusively reserved for time-dependencies and relationships. At present we are writing too much procedure into our problem solutions. Combinatorially, there are many different ways of constructing a flowchart to do the same problem. The variations are limited only by the time-dependencies. That is, A must be computed before B, because A is an input to the computation of B. If, for example, both A and B are input to C, it may not matter to the programmer whether A or B is computed first, but depending upon certain frequency information and other knowledge the compiling routine can well make this decision.

We can look to see (within perhaps two years) an international machine-independent language of the procedure-oriented type which will be suitable for both scientific and commercial work and will be heavily partitioned into organisational entities for the reduction of programmer effort. The processors which accomplish the translation of this language to machine language will be required to be extremely clever and intelligent.

The problem-oriented languages are an upcoming and useful class. In this group we have sort generators (as mentioned before), report generators, file maintenance and updating generators, and table generators. All of these are very highly specialised towards certain frequent and recurring classes of operation. The investiture of the necessary and requisite intelligence into the program is economically justified by the frequency of need.

Let us take the report generator for an example. Input to such a program would be a description of the physical layout of the file, its component structure and the detail structure of these components. The semi-pictorial layout of the output is also required, with indications given of the pagination, margin, number of lines, grouping, spacing, indentation, etc. For a typical report the headings are lettered in exactly as they are to be produced from a typing element in the proper column and row. The working information is laid out exactly as it is desired to be seen with proper decimalisation and auxiliary characters. (Some means of relating this output to the structure of the input file is also necessary.) The cyclical characteristics of data must be specified. It takes a good deal of programming effort to write a good report generator, but there is an extreme pay-off when you invite the vice-president down and hand him an input sheet and say, "Make up your own report." He is shown the simple rules, the information is key-punched and fed to the machine with the working file, and the report comes out in a matter of two or three minutes exactly as that vice-president specified it. It is remarkable how much support a computing installation can get that way.

To finish with machine-independent languages, I should like to emphasise the importance of jargons, and what they do for us. When one considers, for example, the jargons (or dialects) of ALGOL such as NELIAC, CLIP, JOVIAL, MAD, etc., it can be seen that the external appearance of the language is quite a bit a matter of taste. ALGOL reflects certain distinct choices in a matter of exterior form. It has been noted by Julian Green in his work with ALGOL processors that there appears to be a rather rigorous sub-language created from the scan of a string language. This appears to be common regardless of the jargon used. Remarkably enough, it appears to have the quality of Polish Notation with an alternating sequence of operator, operand, operator, operand, etc. This does seem at first to give support to those that prefer Polish notation as the human programming language, as in ADES II and the APT programming language. The group that it actually supports is that which would like to see a specialised jargon for each field of computational need. Mike Barnett, for instance, carries this one step further with his so-called "Macro-directives," which are highly specialised jargons for a particular field. These are translated into an intermediate language such as FORTRAN or ALGOL and then processed into machine language. Brooker has been particularly keen on this, as evidenced by his paper on a self-defined phase-structure language. It would seem that the computer is versatile enough to take specifications of language structure and construct its own rules for translation to the sub-language. Of course, this is directly related to the problem of translation of natural human languages.

Analysis Languages

This is a subject I can touch on only briefly because the field is actually in its infancy, but basically the analysis language should provide the tools to describe the operation of a total system. These are the languages we may expect our systems and procedures analysts of the future to use in describing their problems. There are prerequisites to successful language of this type. Among them are more rigorous methods of describing data organisation and set membership. I imagine they will be much more pictorial, being two- and perhaps three-dimensional. Examples of tabular languages are already in existence, developed by Hunt Foods and by General Electric. In the simplest form the dividing lines between the columns and rows represent and/or conditions. The resulting procedure or operation is described in a column following a double rule. In reality much of this is simply making Boolean algebra more palatable to the user by transformation of the language to a form more compatible with his previous experience. The development committee of CODASYL is extremely concerned with this problem. They point out, and rightly so, that actual programming is often a rather small part the entire analysis problem of today.

Processor Techniques

Two years ago programming was rather in the doldrums. It seemed then that the twenty-five to forty-five man-years necessary to write a major processor were supportable only by manufacturers. Users and universities rebelled at this and so did the manufacturers because of the heavy programming costs. Now we find universities that can write with two man-years of effort better and more sophisticated processors than those which would have required twenty-five man-years as late as 1958. I ascribe this in large part to the development of symbol manipulation techniques.

At an ACM Council meeting a year ago, John Carr was rather perturbed by criticism of ALGOL since he had a large hand in the formation of the effort, and asked "Can anyone tell me just what is wrong with ALGOL?" It fell to me to answer the question and I said, "Simple. It's not a data processing language." In short, ALGOL could not be written in ALGOL. Assembly programs can be written in their own language; why not machine-independent languages? To answer that this is theoretically impossible is wrong. Symbol manipulation is the link. When you are going to ship a language with

< MITCH MILLER & SINGALONG
BOUNCING BALL, RECURSION, GET NEXT CHARACTER

its translator out to face the world so that it can do virtually any problem, you might as well consider one of the most general of these problems. This is the problem of translating from itself to a machine language. In fact, this is the acid test. NEEDED FOR STDS.

When ALGOL came into being as ALGOL 58, we were already embarked upon a language called XTRAN, designed to supplant FORTRAN. Indeed many of the characteristics of ALGOL were born in XTRAN. I asked Julian Green to run an effort to make an experimental processor for ALGOL. He was given only two rules:

1. Nobody that ever worked on a FORTRAN processor was to be associated with the project for fear of prejudice.
2. The processor was to be extremely flexible to accommodate expected changes in ALGOL.

The result of this is an experimental processor still carrying the name XTRAN but capable of providing as many different varieties of ALGOL as one needs. The reason for this is that XTRAN is written in its own language. Symbol manipulation elements have been added. Another successful project of this kind in the United States was undertaken at the System Development Corporation with the languages CLIP and JOVIAL.

As I said in my introduction, most production programmers are unaware of such techniques. The problem is how to convince them to utilise these new techniques. One possible answer lies in a course on compiler construction just given for the first time. This course lasts one week. The first two days are devoted to a special language for symbol manipulation. During the next three days each student writes a complete compiler in this symbol manipulation language and actually checks it out on a machine, in this case the 705. The compiler is a very simple one, and they do not write anything for recursive procedures. Yet it is complete, it works, and is written inside of one week.

Perhaps the second greatest contribution to the programming art in recent years is something we wanted very much to do one or two years ago and only recently discovered how. This is bootstrapping. (I hope the term has the connotation in the United Kingdom as it does in the United States.) In any event, it means to use every possible facility that you have constructed so far in the construction of any new facility. This is not limited to a single machine but may also be extended to moving processors from one machine to another. The most difficult part of bootstrapping is to get that small initial handhold. Normally this starts with handwriting of an origin feature, the assembly of a few instructions, a decoding table for operations and addresses, an assignment feature to actual addresses and a few other such functions. With these facilities one starts to program and moves slowly in an ever-widening circle.

This is the classical method. It was not good enough for Bob Shapiro of the XTRAN project. Shapiro came from the University of Chicago and was not bound by what any other programmer had ever done. He decided that the first tool he needed was a scan to break apart and analyse the elements of the input language. However, he felt that one of the things the scan ought to be able to do was scan itself. So Shapiro wrote what he thought the scan ought to be and then he played machine, ımagining the scan scanning the scan. As he did so, he wrote down the machine instructions that he thought the machine should produce in so doing. He then entered these same machine instructions in the computer and actually fed the scan through the program. In fact, again scanning the scan. This process produced a program for scanning which at first, of course, was not quite the same as that Shapiro had written. He kept at it until the output program in the machine was identical to the program that actually had scanned it. With this he completed his first major bootstrap and saved an enormous amount of work.

Bootstrapping is, however, a more useful device in modest present-day systems. As an example, we were required to produce a processor for a new machine, the 7070. There was a choice between starting from scratch or doing a wasteful job of writing a single translator on another machine—in this case the 705. After some initial opposition I persuaded the production people to write a program in 705 Autocoder which performed the translation from 7070 Autocoder to 7070 machine language. After all, this is a production problem one might be expected to encounter with such a generalised program. The 7070 processor (that is, the processor which would actually run on the 7070) was then written in the full-blown language, taking advantage of every feature available. This was then processed (virtually once and once only) on the 705 to produce a processor which would actually work on the 7070.

The elapsed real time in thus producing the program was greatly reduced, which is very desirable in these days of automatic design and production of machines. We received a bonus we hadn't quite counted on, actually. Now we have one 705 running around the clock, doing nothing but assembling 7070 programs for customers that do not yet have their machines.

XTRAN as an experimental processor has changed form many times, but the basic transformation from independent language to machine language has remained the same. One starts with the scan which produces macro instructions, possibly of a three address nature and quite independent of data configuration. The next step converts these macros to other macro instructions which are data-dependent. For instance, in the original macros we may have been attempting to add a fixed point number to a floating point number or perhaps two fixed point numbers that required decimal alignment, which was not necessary to consider at that time. The next transformation was either to symbolic machine language or direct to machine language through generators. Anatole Holt uses a diagram for this process that I like very much. It is a simple parallelogram which is completely below the base line. This base line represents a dividing position between machine-independent and machine-dependent characteristics. Holt's diagram shows that the transformation is a gradual one through many steps. At each stage there must be a mapping from one form to the other so that no information is lost.

I think this is a good time to dispel the UNCOL myth (Universal Computer Oriented Language.) According to its proponents, all machine-independent languages would translate into UNCOL and UNCOL would be translated to all different machine languages. Apart from the fact that UNCOL has been demonstrated, through the success of CLIP and XTRAN, to be unnecessary, there are certain technical reasons why it cannot exist—excluding if you will the Turing machine. Since UNCOL must comprise the set of all possible machine level operations, it is likely to get outmoded as soon as someone develops a new one. For example, I wonder whether the UNCOL would have included the look-ahead feature of STRETCH if they had designed it five years ago? Then, too, it would seem that to be acceptable to all machines UNCOL would have to translate into the lowest common denominator among all classes of machines and thus the efficiency on each and every object machine would be minimal. I am afraid that as it is presently proposed, UNCOL is a miss, or myth.

To my mind there is an intermediate language form which will serve this same purpose. The only real difference between machine-independent and machine-dependent languages is that they have different constructions reflecting the different organisation of the human mind and the computer mind. To go from one to the other there must be an orderly transmutation of information. I submit that tables and lists can easily be the common denominator for this purpose. Several powerful list processors have already been constructed—LISP of McCarthy and Mealy, and the Newell–Simon–Shaw processors. There are indications from the realm of information storage and retrieval that the day of the list processor has just begun. The ability of various trees to reference recursively both backward and forward on many program levels indicates that they are powerful enough to perform the stated function of UNCOL as an intermediate form. As an example, the XTRAN scan decomposes the string continuously into a matrix. The semicolon as a statement separator is never treated differently from any other character. As a result, arithmetic computations may be optimised over whole sections of the program with redundancies removed. Consider it this way—if one makes a list inside the processor of all the variables that ever have an addition operation performed upon them, it will be detectable that B + A is the same as A + B. All that is required is an ordered list and a search for duplicates.

The translation from a machine-independent to a machine-dependent language raises some interesting speculation. There are two courses open today. One involves translation from the machine-independent language to an intermediate assembly language in machine-like form, with the operators and operands given mnemonic English equivalents. A separate assembly operation then converts this form to machine language. The other alternative is direct generation of machine code. The latter is not enjoying much favour these days. I suspect it will in the future. The proponents of the double step process tell us that machine-independent languages cannot presently state every type of problem, whereas assembly languages can. Therefore, correct machine code in assembly form may be adjoined with the output of the first translation and all translated by the assembly program. This is a safe way to play it, and for today perhaps the most practical for production programming. It is predicated on the assembly and translation processes being long and tedious, such that one could not afford to start over from scratch each time an error is caught or a change made. Direct generation, on the other hand, is based on the principle of recompilation from the beginning each time, although perhaps certain tables of correspondence may be saved. By avoiding the intermediate assembly language step much duplication is avoided and the running program may physically replace the source program in memory.

Another important technique in today's processors is that of flow optimisation. It is well known that there are more devious ways of going to a point four blocks down the street than by walking to it directly. The average programmer left to his own devices is too likely to take many of these detours. The route is best left to the intelligent processor. Perhaps the most complicated section in the various FORTRAN processors is that for flow optimisation through the use of predecessor and successor logic. As you know, the programmer has the option of specifying expected frequency of taking various possible paths at branch points as override information. The processor takes as much of this information as the programmer gives it and constructs a rough test program. Test values of the variables are generated randomly and the test program is exercised with these values to determine any unknown branch frequencies. With this information the program is then reconstructed to optimise the flow such that the most used paths through the program take the shortest time. Of course, if this penalises greatly a slightly less used path, a different choice must be made. Similar to the transportation problem, this technique is in effect a prior optimisation of the program.

Many post-optimisations have been tried with success. This is particularly necessary when we go to macro-instructions to decompose a string language. Normally the macro-instruction generators do not talk to one another. It may well be that the generation of two successive macro-instructions will engender some extraneous commands—multiple store, for example. Other crude rules for optimisation and modification of a program after it has been created fully have been developed.

As one who was brought up on interpretive programs in the early years, it amuses me to see that the compiler is not the last word. To compile implies that you know everything about the program beforehand and all the external characteristics and conditions. In today's multiple processing systems this is definitely not so. Many hardware assignments must be made on-line during actual running. Furthermore, an interpreter is often a more compact form of instruction, whereas a compiler might generate as many as a hundred different ways of doing something, all of which must be maintained in memory in case their particular call should occur. The interpreter effectively generates the proper coding upon demand. The former reason for the unpopularity of interpretive programs was the length of time required for the fetch and interpretation cycles. With proper hardware design, such as that of ATLAS, this is not necessarily a problem.

The interpreter also comes into its own when there is a difference in balance between computational equipment and printing and editing equipment. As a case in point, take a 7090 and a 1401. The 1401 is a small machine with big off-line editing and printing characteristics. To asynchronously operate such equipment on-line with a large machine in a multi-program fashion would require much control information and prior editing. In this case all the 1401 would do would be printing. We have determined that it is very effective for the large machine to construct an interpretive control language as its output, together with the resulting data. The 1401 is nicely able to interpret these control and editing instructions with no loss of printing speed.

A problem of recent interest is the naming facility in processors. I know the English have laughed at some of the three- to five-letter names one encounters in American programming systems. I admit this is quite unnecessary and I apologise. The possible names one could use of any number of characters form a very sparse set. It is very expensive to carry around character by character representations in the compiling and translating process. These names are meaningful only to the programmer. They may be exchanged for compact binary representations for use in machine processing. A double list of these relationships is maintained for availability whenever output is required.

The problem of locating files by their names is related to this. With random access memory it is cheaper and more convenient to transform the name into a unique address which locates the related file rather than perform a special table search for the name and find the associated address. Lists come into their own here, and chaining techniques have been developed. That is, one converts the numeric representation of the name into a more compact number. In the address given by this number one should find the original name to serve as verification. If not, a chaining address is also given for the next try. The need for this is occasioned because the conversion algorithms sometimes produce duplicates in a more dense set. However, the expenditure of search time is far, far less than that for binary search. On typical files where 20% of the total files get 80% of the activity, the average number of searches made in a fully packed file has been determined as 1·12. Operating in this fashion is also good practice for the days of associative memory.

It was a combination of this chaining technique, the work of Shannon, and zero-compression techniques that led to the development of "Digital Shorthand" as a communications code. With computers on each end of a communications line, rather than the simple terminal equipments of today, we can transmit three times the volume of formatted text in compressed form, decompressing it at the receiving end. Facsimile may be sent at a saving of 4 to 1. This method promises large savings over expensive communications linkages such as Atlantic cables and satellites. An experimental 7090 program indicates that, with full utilisation, the cost of both sending and receiving computers is about 0·006 pence per word. Contrast this with 1s. 6d. per word day rate, London to New York, or 9d. night rates. This scheme will handle the full English dictionary at an average of 10·7 bits per word. LANG TRAN WITH PRIOR KNOWLEDGE

I have briefly touched on some of the more salient features and techniques that make large gains in both the writing of processors and the running of the programs they produce. Now to move to my final topic, the one probably dearest to my heart, that of operating systems.

Operating Systems (c̄ LANG)

There has been a steady trend away from the combined human-machine operation and toward fully automatic machine-controlled operation. There is no doubt but what the vast increases in machine speed have forced this, but it would have been a desirable development even if speeds had remained the same. The first large automatic operating system, developed at General Motors for the 704, doubled the working efficiency of that machine.

One of the most important components of an operating system is the IOCS, which stands for Input/Output Control System. The proper scheduling of Input/Output is a far more difficult matter than writing the procedure. With IOCS we see new verbs introduced such as GET, PUT, INTERLOCK, OPEN and CLOSE FILE. All of these are compound instructions generated for maximum efficiency in feeding data to the operating procedures for producing answers.

Obviously a complex system of this nature has many levels of operation. Control must exist through hierarchies of overrides and limits. All component functions must be organised as subroutines eventually called by the topmost level of control. Since the scheduling function is one of these components there must be access to all machine states by interrogation or trapping. If trapping is used it must be capable of being disabled and enabled by the control program.

The scheduling function may be primitive or very complex. A good deal of development is being done by Codd and Held in the United States. Until a radical change in machine design, however, I am inclined to favour the primitive approach for a sensible profit;

HOW OSCAR WOULD HAVE LOVED THIS!
650 IS DRUM, APDM,

too many experimental scheduling programs now take up more time in making the decision than the machine time they gain.

Assignment of operating units must not be made in the program proper. This is left to the operating system, which makes real time assignment according to what it has available soonest. For a tape unit, for example, this is probably the first unit the previous running program has relinquished. The programmer must in general refer to physical units by abstract names. This may be carried to the point of random loading of tape units. At the beginning of each problem the control program reads the labels on each tape unit to find out what exists there. It may also interrogate memory to find out how much is available and adjust the program accordingly for more efficiency. Self-adjustment to machine configuration is not costly for such a powerful device.

We would expect the program in the original language to be stored somewhere for ready access. Self-repair of programs may be effected by returning to the more compact source form. This is connected to the self-repair of the machine itself. A diagnostic program contained in the operating system may be called upon to test for faulty machine elements. Upon discovery, a message would be typed out to the service man, but rather than halt operations, either the current program would be readjusted by partial recompilation to avoid the faulty area, or another program might be started which did not require it.

Experiments indicate the possibility of successful diagnosis on a time-shared basis. This enables the programmer to essentially talk to the machine in real time at his convenience. Of course, all diagnosis is done and results obtained in the machine-independent language the program was originally written in.

Once a self-operating system is postulated and begun, no matter how primitively, we are on the way to remote shared operation of very large machines. The graphs of problems per monetary unit always show remarkable decreases when the machine gets larger and faster. I have long envisioned computers larger than STRETCH acting as large service and message centres. Because they must be tied in with communications networks for this purpose, they are automatically available for message control and forwarding, text and facsimile compression to high efficiency and low cost, and a variety of related functions. Certainly the very organisation in this manner will more than amortise the cost of the computers.

This concept would indicate that vast files of read-only memory will be an important requirement for the future. Even program instructions may be largely fixed and unalterable. Old-time programmers remember a lot of instruction modification, but how much do you need now with index registers, indirect addressing to many levels and symbolic addressing? I would venture to say that less than 10% of our program instructions ever get modified now, and the percentage will become much less.

I thank you for this wonderful opportunity to address you, and if you think that I have been talking too much "futures," read a copy of this talk three years from now and see how old-fashioned the ideas are.

UNWIN BROTHERS LIMITED, WOKING AND LONDON

Techniques

Editor's Note

I have seen a number of subroutines lately for the computation of transcendental functions. To a routine, the range of the argument is at least as high as $\pi/4$. Nobody seems to use tables to break up the range, nor do they make use of trigonometric identities.

I remember very well the day Al Podvin's sine-cosine routine for the 650 ran faster than mine because he had used the formula for sine 3θ. For the information of subroutine planners who have not investigated this technique, Table I gives some alternate expressions for the computation of sine and cosine of $n\theta$. The clever programmer will now see how he can trade off extra computation here for reduction in the size of the approximating polynomials due to reduced range. Telescoping the coefficients is also advisable.

I believe tricks like this will be highly efficient because of the large proportion of applications which demand the computation of *both* sine and cosine; theodolite data reduction, for example. For binary machine users, it may be of interest to notice how many of the coefficients in Table I are susceptible to shifting techniques for multiplication.

Let me anticipate the reader's complaint that θ must be computed with more precision. Do not divide the argument, $n\theta$, by n, but rather multiply by the reciprocal of n. This leads to another interesting (and apparently little known) characteristic of binary machines The reciprocals of all decimal integers are of course repeating fractions, whether expressed in decimal or binary. However, certain of these integers have the property that the binary repeating cycle is very short. Table II is therefore of interest. The quantity enclosed in parentheses repeats continuously.

The alert programmer will now see ways to perform certain multiple-precision divisions by integers in less time than the standard machine instructions would take, by taking advantage of masking and shift instructions.

R. W. B.

TABLE I. *Formulae for calculating* sin $n\theta$, cos $n\theta$

n	Sin θn	Cos $n\theta$
1	S	C
2	$2SC$	$C^2 - S^2$ $1 - 2S^2$ $2C^2 - 1$
3	$3S - 4S^3$ $S(3 - 4S^2)$ $S(4C^2 - 1)$	$4C^3 - 3C$ $-C(3 - 4C^2)$ $-C(4S^2 - 1)$
4	$4SC(1 - 2S^2)$ $4SC(2C^2 - 1)$ $4SC(C^2 - S^2)$	$4(S^4 + C^4) - 3$ $1 - 8C^2 + 8C^4$ $1 - 8S^2 + 8S^4$ $1 - 8C^2S^2$
5	$5S - 20S^3 + 16S^5$ $S(5 - 20S^2 + 16S^4)$ $S(1 - 12C^2 + 16C^4)$ $S(4C^2 + K_1)(4C^2 + K_2)$	$5C - 20C^3 + 16C^5$ $C(5 - 20C^2 + 16C^4)$ $C(1 - 12S^2 + 16S^4)$ $C(4C^2 + K_3)(4C^2 + K_4)$

26

TABLE II. *Binary reciprocals of some small decimal integers*

N_{dec}	1/N Binary	N_{dec}	1/N Binary	N_{dec}	1/N Binary	N_{dec}	1/N Binary
1		26	.0(000100111011)	51	.(00000101)	76	
2	.1(0)	27		52		77	
3	.(01)	28	.00(001)	53		78	
4	.01(0)	29		54		79	
5	.(0011)	30	.C(0001)	55		80	.0000(0011)
6	.0(01)	31	.(00001)	56	.000(001)	81	
7	.(001)	32	.00001(0)	57		82	
8	.001(0)	33	.(00C0011111)	58		83	
9	.(00C111)	34	.0(0000C1111)	59		84	.00(000011)
10	.0(0011)	35	.(00000111C1C1)	60	.00(0001)	85	.(0000C011)
11	.(0001011101)	36	.00(000111)	61		86	
12	.00(01)	37		62	.0(00001)	87	
13	.(000100111011)	38		63	.(000001)	88	
14	.0(001)	39	.(0000C1101001)	64	.000001(0)	89	
15	.(0001)	40	.000(0011)	65	.(000000111111)	90	
16	.0001(0)	41		66	.0(0000011111)	91	
17	.(00001111)	42	.0(000011)	67		92	
18	.0(000111)	43	.(00000101111101)	68	.00(00001111)	93	
19		44	.00(0001011101)	69		94	
20	.00(0011)	45	.(000001011011)	70		95	
21	.(000011)	46	.0(00001011001)	71		96	.00000(01)
22	.0(0001011101)	47		72	.000(000111)	97	
23	.(00001011001)	48	.0000(01)	73	.(000000111)	98	
24	.000(01)	49		74		99	
25		50		75		100	

1973 June 14

Mr. J. Robert Logan
Data Systems Division
Litton Systems, Inc.
Van Nuys, CA

Dear Mr. Logan:

For historical interest, in connection with your article "Designing a Binary Reciprocator", in the 1973 May issue of Computer Design, I attach a copy of my Editor's Note in the 1961 April issue of the Communications of the ACM. My Table II shows the reciprocals of your Table I, except that the repeating element is also indicated.

Isn't it strange that these things get lost when new technologies come along? I am pleased that you have resurrected this usage after twelve years.

Sincerely yours,

R. W. Bemer

n

cc: Sydney Shapiro, Managing Editor

Standards

EDITOR'S NOTE

This paper describes a tentative standard code which has been proposed to the membership of the ASA X3.2 Subcommittee on Character Sets and Input/Output Media. The X3.2 Subcommittee is now examining this work in detail and, in my opinion, will probably recommend as an American Standard a code which is quite similar in principle to the one described here although it may differ in some details. Because of the urgency of this work and the very great amount of effort which has already gone into the development of this proposal, it is presented here, informally, for the information and scrutiny of the ACM membership although at proof time it had not yet been reviewed by the ACM Standards Committee for consideration as an ACM Standard. The authors and this Department will welcome your comments.

*Design of an Improved** Transmission/Data Processing Code

R. W. Bemer, H. J. Smith, Jr., F. A. Williams, Jr.
IBM Corp., White Plains, N. Y.

Historically there has been strong difference of opinion in the construction of 6-bit (64-character) data codes, based upon whether the code is to be used for communications or data processing. This paper reports on investigation of an improved code which meets transmission requirements and requires very little modification for varied data processing usage.

It has been evident from the workings of the ASA Subcommittee X3.2 that the transmission people are not as adaptable to modifications as the data processing people. This is simply a matter of inflexibility of existing semi-mechanical communications equipment compared to the general-purpose nature of electronic data processing equipment.

The major obstacle lies in the collating, or ranking, sequence of the characters of the set. It is true that a large proportion of the ordered files of today are sequenced on numeric keys alone. However, a substantial proportion of these files are ordered on keys which contain alphabetic and special punctuation characters. If a standard code changes the relative ranking of such characters the presently ordered files will all have to be fully reordered to the new sequence, a process requiring a great expenditure of machine time. Transformation of one bit representation to another is relatively simple when the sequencing property is ignored. However, one should try to guarantee that the files are still in proper order after such conversion [1].

There are three inputs to the collating problem:

(1) The most prevalent ranking in the U.S. is that established by IBM equipment, particularly the 705. In order are the blank, special characters, the alphabet, the digits. The critical point here is that the digits are higher than the alphabet, for whatever reason. The United Kingdom and certain other U.S. manufacturers (Sperry Rand and RCA) rank the digits lower than the alphabet.

(2) The desire of communications people, as first evidenced by Fieldata [2, 3, 4], is to have the 6-bit set collapsible to a 5-bit Baudot-type set with effectively the same characters. This is to utilize existing Baudot-Teletype equipment with simple modification.

(3) Certain punctuation characters, by universally accepted practice, should collate low to alphabets, digits, and other special characters. For example, the following two names would normally be ordered:

Roberts, A. B.
Robertson, X.

whereas the Fieldata code, because the comma ranks higher than the alphabet, would yield an ordering:

Robertson, X.
Roberts, A. B.

Expansion and contraction between any of the 4-, 5-, 6-, 7- and 8-bit code sets demand a certain uniformity and simplicity. Thus the alphabet should be reserved to two contiguous quadrants of the four quadrants of the 6-bit set. The choice now appears as in Figure A.

* Revision, 15 Mar. 1961.

	Quadrant			
(TYPIFIED BY)	1	2	3	4
Bendix G-20, GAMMA 60	Alphabet	Alphabet	Digits	Special
Fieldata	Alphabet	Alphabet	Special	Digits, special
IBM Stretch	Blank, special	Alphabet	Alphabet	Digits
U.K. [5]	Blank, special	Digits	Alphabet	Alphabet

FIG. A

In the opinion of the authors neither the Fieldata code nor the U.K. code meet the criterion for 5-bit Baudot-like operation completely, even though that was one of the major design requirements. A Baudot type of code is formed essentially as follows:

Letters	Control
Digits and special	

32 5-bit combinations

In any 2-mode code for paper tape, three of the control codes, DELETE, FIGURE SHIFT and LETTER SHIFT, *must* invariably be common to both shifts. DELETE must be all 1's (all punched on paper tape) and MASTER SPACE must be all 0's (unpunched tape). MASTER SPACE, BLANK, and ESCAPE preferably appear in both shifts. Such controls as LINE FEED, CARRIAGE RETURN need appear in only one shift, but operation is more complicated.

Some of these functions may be combined in a single code combination. DELETE/LETTER SHIFT is a single code in Baudot. FIGURE SHIFT is synonymous with one of the three functions possible to ESCAPE. [6]

Since DL, FS and LS must be common to both modes, Fieldata loses the Y and Z of the alphabet and the − and + characters in the collapsed 5-bit mode. This is not tolerable because some words are spelled using Y and Z. Similarly, the U.K. code loses the letters F and G and the symbols . and −. The code developed in this paper is very similar to both of these codes but removes these major flaws.

All of the criteria of the Fieldata study are used here. The full spectrum of expansion and contraction among 4-, 5-, 6-, 7- and 8-bit sets is considered in addition. Thus there are the following additional criteria and remarks:

1. A collating sequence has utility in data processing codes containing alphabets; transmission codes do not require such a sequence.
2. A collating sequence has no utility in a 4-bit set.
3. A collating sequence has utility in 5- and 6-bit sets and it is desirable that the sequence correspond to the binary representations.
4. If it is assumed that the 7- and 8-bit sets contain upper and lower case forms of the same alphabet, it is impossible to have the collating sequence match the binary representations, for the case distinction is of lesser significance than the distinction between characters with different meanings. [7, 8]
5. It is not necessary that the full 4-bit set be in 16 contiguous positions in larger sets. It is only necessary that the lowest four bit positions form the dense, unduplicated set. Other bit positions may vary. However, the digits 0-9 (10, 11) should be certainly be grouped contiguously in any set.
6. Punctuation characters have natural delimiting functions and should thus collate low to both the alphabet and digits. These include, but are not limited to:

   blank . , / − : ; ′ ( ) (not in ranked order)
7. Since period and hyphen are natural delimiters, they should be placed low to both alphabets and digits. However, they often serve as radix point and minus sign (which are not delimiters) in the 4-bit numeric set. There must also be a character in this set to serve as a blank; this may or may not print in the 4-bit numeric mode. Therefore any characters of the 4-bit set which are delimiters should be in a different contiguous block than the digits, so they can serve the delimiting function in larger sets. There should be some regular transformation to append bits when expanding to larger sets.
8. All expansion and contraction from and to the various set sizes shall be blind, without knowledge of the meaning of the character assigned to any bit representation, or of contextual adjacency (with the exception of FIGURE/LETTER SHIFT control in going between 5- and 6-bit sets).
9. In all expansion and contraction, MASTER SPACE must remain all 0's and DELETE must remain all 1's. ESCAPE shall always be the second highest code, one less than DELETE; thus all bits except the low order are 1. For paper tape usage, BLANK must be different from MASTER SPACE and therefore shall have all bits 0 except the low order. This guarantees that BLANK, as the primary delimiter, collates low to all other characters. It is also the complement of ESCAPE.
10. All possible caution should be exercised in alphabetic regions to provide maximum expansion for non-English alphabets (> 26 letters).

The 8 bits are represented by B_7 through B_0, high to low order. The 6-bit transmission set will be developed first. Figure 1 shows a modified Fieldata pattern with B_5 not yet assigned, reflecting criterion 9 only. $B_5 = 0$ for Fieldata, $B_5 = 1$ for U.K.

It is now obvious that LS and FS should be opposite ES and DL, not MS and BLANK, in order to maximize the number of punctuation characters following BLANK. Since the decimal digits must have their binary representation equal to the binary value, they must be placed

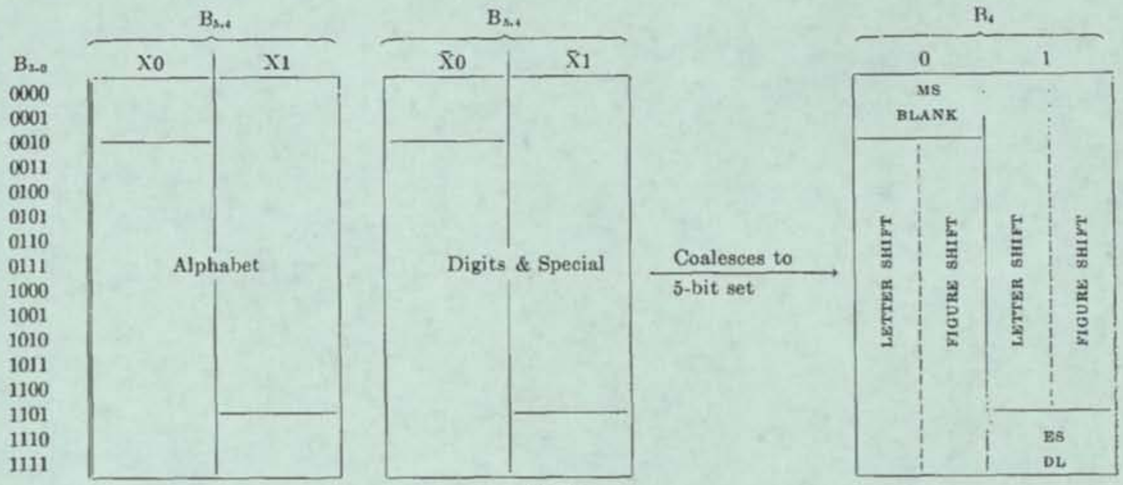


Fig. 1

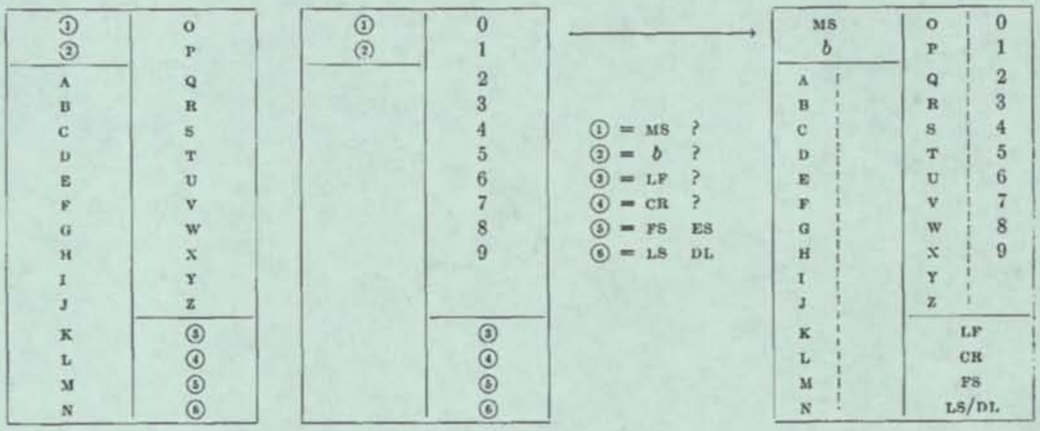


Fig. 2

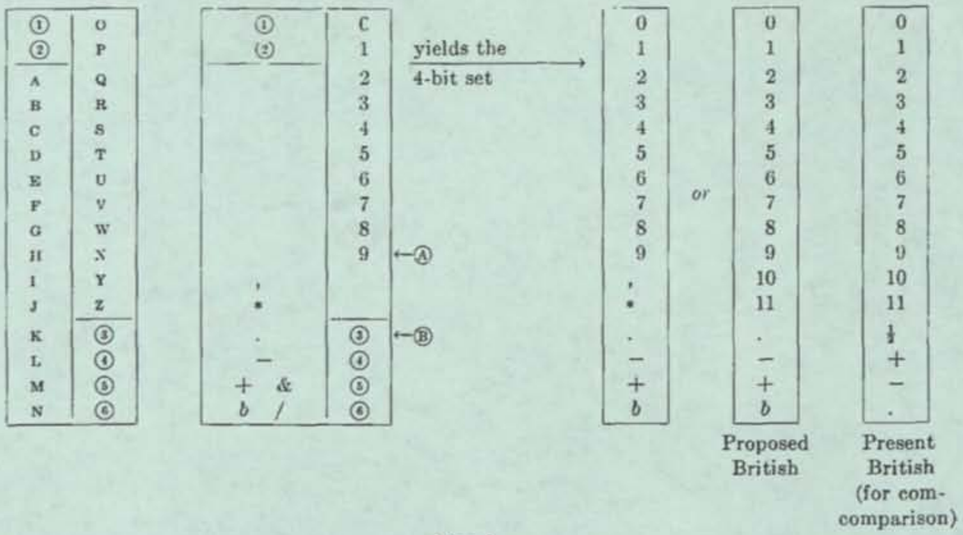


Fig. 3

For	.	−	+ &	/	,	(	)	'	:	?	=	$	# £	;	"	!	•	@	□	%	<	>
Basic CCITT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓											
Teletype	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓						
Basic IBM Printers	✓	✓	✓	✓	✓							✓	✓				✓	✓	✓	✓		
FORTRAN Printers	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓					✓					
1410, 7070, COBOL	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓			✓	✓	✓	✓	✓	✓

FIG. 4

in the $\bar{X}1$ quadrant. This is shown in Figure 2. LINE FEED (LF) and CARRIAGE RETURN (CR) are necessary for character-at-a-time[1] printers associated with existing communication systems. As control signals, they are grouped with other control signals rather than with MS and BLANK, which are essentially informational.

There is space for 20 special characters in the 6-bit set, but four of these must disappear in the 5-bit set.

In conformity to most existing practice, the other six characters of the 4-bit set have been selected as:

. − + (for self-delimiting data fields)
ƀ printing separator (may have graphic representation)
, digit grouping
* indicator for totals, etc.
most expendable for British pence (10, 11)

The two positions following the digit 9 are not usable for delimiters in the 6-bit set, since they will collate high. These six characters are assigned as shown in Figure 3. The pairs (.,) and (−+) are a distance of two bits apart, for easier error detection. + is used fully interchangeably with &, since & may also take the forms ✢ ₰ &. / has been chosen as alternate for the BLANK in the 4-bit set, since it can serve very well as a space indicator, as 00/636/505///21.

$B_{3\text{-}0}$	$B_{5,4}$			
	00	01	10	11
0000	MS	0	①	O
0001	ƀ	1	②	P
0010	"	2	A	Q
0011	$	3	B	R
0100	#	4	C	S
0101	'	5	D	T
0110	(	6	E	U
0111	)	7	F	V
1000	:	8	G	W
1001	;	9	H	X
1010	,	?	I	Y
1011	•	!	J	Z
1100	.	LF	K	③
1101	−	CR	L	④
1110	+	FS	M	ES
1111	/	LS	N	DL

Transmission Code

$$\xrightarrow[B_4' = B_4 \forall B_5]{B_5' = B_4}$$

$$\xleftarrow[B_5 = B_4' \forall B_5']{B_4 = B_5'}$$

$B_{5,4}$			
00	01	10	11
NULL	<	O	0
ƀ	>	P	1
"	A	Q	2
$	B	R	3
#	C	S	4
'	D	T	5
(	E	U	6
)	F	V	7
:	G	W	8
;	H	X	9
,	I	Y	?
•	J	Z	!
.	K	=	CONTROL
−	L	@	
+	M	%	
/	N	□	

Data Processing Code

FIG. 5. The Proposed Standard Code

The basic set is achieved by changing the transform at Ⓐ.

$$B_4 = \overline{B}_3 \vee (\overline{B}_2 \wedge \overline{B}_1)$$

The British set *should* have 10 and 11 immediately following 9. This is achieved by changing the transform at Ⓑ.

$$B_4 = \overline{B}_3 \vee \overline{B}_2$$

Figure 4 gives the special characters specified in existing systems.

A FORTRAN-commercial substitution exists to overcome limited capacity of line printers. The correspondence is:

to = @ to ' % to (□ to)

The Bell and "who are you" functions are ignored here because they do not warrant individual characters. They are handled best by the ESCAPE mode.

B_5 may now be assigned specifically.

$X = 0,\ \overline{X} = 1$ yields a modified FIELDATA which, unless transformed, has punctuation high to the alphabet. This is not logically consistent.
$X = 1,\ \overline{X} = 0$ yields modified U.K.
" UNIVAC, MH
" RCA 501
" 704 internal

We will thus choose the latter. This choice also diminishes the number of bits or punches in numeric data, which is most frequent to data processing.

The specific proposal of Figure 5 implies either that:

(a) the data processing code is internal, and the EXCLUSIVE OR mapping takes place at the interface on reading or writing externally on media such as tape or communication lines, which utilize the transmission code, or

(b) the data processing code is merely figurative and represents the effective collating sequence obtained by a simple comparison logic in the machine.

The transmission code folds to the Baudot-like code of Figure 6. It retains all the special characters of present-day Teletype, plus the *. Although ? and !, as effective delimiters, might well precede the alphabet in the data processing code (involving a swap with < and >), to do so would remove ? and ! from the 5-bit transmission code. If the transmission people agree, this change could be considered.

As a 6-bit transmission code, ① to ④ are available. These might be used either for additional control func-

[1] The common term in communications is "page" printer; however, printers which print an entire page at one time must preempt this term.

MS		O	0
b		P	1
A	"	Q	2
B	$	R	3
C	#	S	4
D	'	T	5
E	(	U	6
F	)	V	7
G	:	W	8
H	;	X	9
I	,	Y	?
J	*	Z	!
K	.	LF	
L	—	CR	
M	+	FS	
N	/	LS/DL	

Fig. 6

	<	O	0
	>	P	1
	A	Q	2
$	B	R	3
#	C	S	4
'	D	T	5
(	E	U	6
)	F	V	7
:	G	W	8
;	H	X	9
,	I	Y	?
*	J	Z	!
.	K	=	
—	L	@	
+	M	%	
/	N	□	

57 + Blank

		O	0
		P	1
	A	Q	2
$	B	R	3
C #	C	S	4
F '	D	T	5
F (	E	U	6
F)	F	V	7
	G	W	8
	H	X	9
,	I	Y	
*	J	Z	
.	K	F =	
—	L	C @	
+	M	C %	
/	N	C □	

48 + Blank

Fig. 7

(Identical to Fig. 6)

MS		O	0
b		P	1
A	"	Q	2
B	$	R	3
C	#	S	4
D	'	T	5
E	(	U	6
F	)	V	7
G	:	W	8
H	;	X	9
I	,	Y	?
J	*	Z	!
K	.	LF	
L	—	CR	
M	+	FS	
N	/	LS/DL	

5-bit

$\xrightarrow{\forall}$ $\xleftarrow{\forall}$

MS	①	O	0
b	②	P	1
"	A	Q	2
$	B	R	3
#	C	S	4
'	D	T	5
(	E	U	6
)	F	V	7
:	G	W	8
;	H	X	9
,	I	Y	?
*	J	Z	!
.	K	③	LF
—	L	④	CR
+	M	FS	ES
/	N	LS	DL

6-bit

Transmission Codes

→ ←

(Identical to Fig. 7)

NULL	<	O	0
b	>	P	1
"	A	Q	2
$	B	R	3
#	C	S	4
'	D	T	5
(	E	U	6
)	F	V	7
:	G	W	8
;	H	X	9
,	I	Y	?
*	J	Z	!
.	K	=	CONTROL
—	L	@	
+	M	%	
/	N	□	

Data Processing Code

Fig. 8

tions or (preferably) for additional characters of foreign alphabets. FS and LS are also available, once the complete change is made from existing equipment. Fieldata would then use these characters as UPPER CASE (UC) and LOWER CASE (LC) respectively. This provides a representation of a 7-bit code in 6-bit form, just as Baudot represents a 6-bit code in 5-bit form. Therefore the FS-UC and LS-LC correspondences are true, and either mnemonic might be used. Perhaps a new combination would be desirable, as FU (for Figure/Upper) and LL (for Letter/Lower).

= @ % and □ are placed high in the data processing code, and it is assumed they will not be used in control keys. Figure 7 shows the DP code satisfying the last set of special characters of Figure 4, plus the FORTRAN transformation and a 48-character set.

The proposed set has the special characters assigned for reasons other than matching correspondence between the digits and the characters associated with those digits on typewriter keyboards. The reasons are:

1. Some typewriters do not have keys for the digit one (1) or even for the digit (0).

2. There is no such thing as a standard typewriter keyboard in the U.S. There is a proposed British Standard, but the characters placed most uniformly, the left and right parentheses, are above 9 and 0 respectively. This conforms with much practice in the U.S., but 0 must be placed in parallel to MS in accordance with our previous rules.

3. Transmission people sometimes desire the parentheses over the 8 and 9 respectively, but this occurs only in the Luebbert revision of Fieldata (not the original Fieldata) and Ferranti computers.

4. Users normally adjust automatically to any arrangement of special characters after a day's usage.

5. Non-English keyboards differ greatly in this placement. It would be unfair to inflict one of the many English arrangements as an international standard.

6. The FORTRAN-Commercial interchange is accomplished in the proposed set by recognizing

$$B_2 \wedge \overline{B}_4 \wedge (B_3 \equiv B_5),$$

and inverting B_3 and B_5 if this condition is true. Any other arrangement greatly complicates the logical hardware necessary in converting existing printers, probably a more expensive process than converting existing Teletypewriters.

7. Any such correspondence will still require two modes of keyboard logic to generate codes.

If the transmission people could modify existing equipment with an EXCLUSIVE OR function (two relays), a completely common and collatable 6-bit code could exist, as shown in Figure 8, subject to the requirements for expansion to 7- and 8-bit sets.

REFERENCES

1. Electronic Industries Association, Basic Character Set Code, Tentative Standards Proposal 7233, May 1960.
2. LUEBBERT, W. F. Information handling and processing in large communication systems. Tech. Report 099-1, Stanford Electronics Laboratories, Stanford University, 11 July 1960.
3. U. S. Army Signal Corps, Fieldata Equipment Intercommunication Characteristics. 1 August 1959.
4. LUEBBERT, W. F. The design of the new military common-language data code. (Dittoed Copy).
5. British Standards Institution, Committee DPE 149, Draft British Standard for Punched Tape Coding, Part 1, 7 Track Code, AA (DPE) 3543, Sept. 1960.
6. BEMER, R. W. A proposal for character code compatibility. *Comm. ACM 3*, No. 2, Feb. 1960.
7. BEMER, R. W. On the design of extended character sets. 26 Jan. 1961 (Unpublished).
8. SMITH, H. F. JR. AND WILLIAMS, F. A. JR. Remarks on collating sequences. 22 Sept. 1960 (unpublished).

APPENDIX

On the Relative Position of the Alphabet, Numbers and Special Characters in a Code Set Based upon Transmission and Data Processing Characteristics

Consider a 4-quadrant arrangement of 16 states per quadrant in which Q1, Q2, Q3 and Q4 represent the octal codes 00-17, 20-37, 40-57 and 60-77. Consider also four classes of information symbols which may be placed in these quadrants:

D representing the 10 or 12 digits of the decimal or duodecimal system
S representing the class of special characters
A representing a section of the alphabet beginning with the letter A
Z representing a section of the alphabet ending with the letter Z

If each of these four classes of information is assumed to consist of up to 16 codes, they may be assigned to the 4 quadrants in any of 24 combinations. We now consider these combinations in light of their desirability for data processing and data transmission.

Transmission Considerations

1. Concepts of MASTER SPACE and BLANK are distinct. The term "BLANK" refers to the element of information used to separate words on a printed page. MASTER SPACE occupies the zeroth position.
2. The concept of ERASE or DELETE is represented by the *N*th character.
3. MASTER SPACE, BLANK and DELETE are concepts required in all alphabetic or alphanumeric sets. Further MS and DL always occupy the same relative position in each set.
4. The 64-state code set is to be representable by a 32-state code using a shift mode. In this compressed representation the alphabet is to form one shift and the numbers and special characters the other.

These criteria imply:

01. A and Z cannot fold upon each other (4).
02. Q1 and Q4 cannot fold upon each other (1, 2, 3).
03. MASTER SPACE must be in Q1. (1)

Data Processing Considerations

5. The digits are represented by their pure, natural binary equivalents. Since only four bit positions are necessary to represent up to 16 states, any additional bit position in a given set must contain the same pattern of 1's and 0's for each digit.
6. No symbol other than MASTER SPACE ranks lower than BLANK in the collating sequence.
7. The alphabet is dense in collation.
8. Certain field-separating symbols including BLANK must rank lower than the alphabet in collating.

These criteria imply:

04. D cannot be in the quadrant which contains or folds on MASTER SPACE. Otherwise 0 and MASTER SPACE would become identical in some code set. (5)
05. Z cannot occupy Q1. (7)
06. MASTER SPACE and BLANK must appear as adjacent characters in the same quadrant. Otherwise some symbols will be less than BLANK or MS will have a rank higher than BLANK. (6)
07. BLANK cannot be associated with A. Otherwise some special symbols would either be lower in collating than BLANK, or field-breaking symbols would be higher in collating than the alphabet which they are intended to separate. (6, 7)
08. (7) and (8) immediately above imply that A cannot occupy Q1 since MS must be located here.

Application of Rules to the Combinations of D, A, Z, and S on Q1, Q2, Q3, and Q4

The requirements 04, 08 and 05 remove from consideration the 18 combinations beginning with D, A and Z respectively. In addition any combination in which A does not precede Z can result in a non-dense alphabet.

(*please turn to page 225*)

TABLE 1

Deviates of the Normal Function in Octal Corresponding to the Cumulative Area From .5 to 1.0

Area (Octal) Scaled 2^8	Normal Deviate (Octal) Scaled 2^{28}	Area (Octal) Scaled 2^8	Normal Deviate (Octal) Scaled 2^{28}
200	00 00000 00000.	300	00 12625 32704.
	00 00120 15457.		00 12772 42473.
	00 00240 33636.		00 13140 41660.
	00 00360 53002.		00 13307 32316.
	00 00500 73431.		00 13457 16144.
	00 00621 16043.		00 13627 77435.
	00 00741 43141.		00 14001 60730.
	00 01061 73214.		00 14154 44377.
210	00 01202 26537.	310	00 14330 34471.
	00 01322 66042.		00 14505 34031.
	00 01443 32032.		00 14663 45573.
	00 01564 03006.		00 15042 74314.
	00 01704 61253.		00 15223 42707.
	00 02025 45515.		00 15405 34463.
	00 02146 40500.		00 15570 55071.
	00 02267 42513.		00 15755 27637.
220	00 02410 54064.	320	00 16143 40227.
	00 02531 75526.		00 16333 12607.
	00 02653 30005.		00 16524 33524.
	00 02774 73427.		00 16717 27240.
	00 03116 50535.		00 17114 02241.
	00 03240 40077.		00 17312 41757.
	00 03362 42443.		00 17512 73747.
	00 03504 60346.		00 17715 25671.
230	00 03627 12161.	330	00 20121 65564.
	00 03751 60466.		00 20330 42335.
	00 04074 44256.		00 20541 43267.
	00 04217 45711.		00 20755 00012.
	00 04342 65610.		00 21173 00365.
	00 04466 24554.		00 21413 55747.
	00 04612 03410.		00 21637 22316.
	00 04736 02547.		00 22065 70106.
240	00 05062 22626.	340	00 22317 52300.
	00 05206 64474.		00 22554 65705.
	00 05333 51003.		00 23015 50463.
	00 05460 60415.		00 23262 21226.
	00 05606 13615.		00 23533 00141.
	00 05733 73512.		00 24010 07632.
	00 06062 01012.		00 24271 74500.
	00 06210 34430.		00 24560 64671.
250	00 06337 16705.	350	00 25055 11231.
	00 06466 30775.		00 25357 26315.
	00 06615 73671.		00 25667 74207.
	00 06745 70150.		00 26207 36451.
	00 07076 16612.		00 26536 46016.
	00 07227 00465.		00 27076 02014.
	00 07360 16636.		00 27446 51276.
	00 07511 72144.		00 30031 33024.
260	00 07644 03467.	360	00 30427 40773.
	00 07776 54160.		00 31042 23633.
	00 10131 65174.		00 31473 35502.
	00 10265 37466.		00 32144 76235.
	00 10421 54234.		00 32641 23053.
	00 10556 34507.		00 33363 41636.
	00 10713 61765.		00 34137 46520.
	00 11051 55313.		00 34752 51075.
270	00 11210 20025.	370	00 35633 37754.
	00 11347 33317.		00 36573 54572.
	00 11507 21011.		00 37631 00460.
	00 11647 62131.		00 41010 33357.
	00 12011 00143.		00 42354 42517.
	00 12152 74634.		00 44204 72526.
	00 12315 51606.		00 46534 51133.
	00 12461 10461.		00 52437 23555.

STANDARDS—Continued from page 217:

As can be seen, this is not the only method of applying the stated rules. Despite the order taken the rules reduce to three the number of acceptable combinations for data processing and transmission. These combinations are S, A, Z, D which is followed by IBM and S, D, A, Z which is advocated in the United Kingdom. Also possible is S, A, D, Z.

The FIELDATA arrangement A, Z, S, D is not acceptable. First, BLANK is associated with A, which means the delimiting special characters will collate higher than the alphabet. Second, if BLANK is not associated with MASTER SPACE in Q1 but is in the second position of Q3, one symbol (the character in the first position of Q3) other than MASTER SPACE must have a rank less than BLANK.

This examination of the possible combinations of S, D, A, Z merely indicates which arrangements should be further investigated with view of their expansion characteristics in sets of more than 6 bits. The analysis is intended to remove much of the confusion which has existed as to what combinations are possible and desirable for expansion and contraction. Thus, the arrangements S, A, Z, D and S, D, A, Z will be given further analysis and a choice between them made on the facility of their expansion and contraction characteristics.

Folding Considerations

At this point the methods of folding must be considered. The four quadrants may be folded in one of the two ways by the removal of either B5 or B4.

Q1 on Q2 and Q3 on Q4
Q1 on Q3 and Q2 on Q4

When the folding consideration is applied to the remaining combinations of S, A, D, Z we find,

	Q1 on Q2	Q1 on Q3
SDAZ	Note 1	Possible
SAZD	Note 2	Note 4
SADZ	Note 3	Note 1

NOTE 1. The digits cannot fold on MASTER SPACE.

NOTE 2. The transformation characteristic between 5 and 6 bits is dependent upon the combination being treated as well as the shift.

NOTE 3. This is possible by treating B_4 of the 6-bit representation as the mode bit. However, this leads to a non-dense alphabet.

NOTE 4. Z cannot fold on MASTER SPACE.

Conclusion

From the consideration of data transmission and data processing criteria we are led to a code organization of S, D, A, Z. This organization, however, should not be considered as giving the collating sequence.

29

Sonderdruck aus
Reprinted from

Digitale Informationswandler

Probleme der Informationsverarbeitung in ausgewählten Beiträgen

Selected Articles on Problems of Information Processing

Herausgeber/Editor
Walter Hoffmann

Autoren/Contributors

Yehoshua Bar-Hillel
Jerusalem
Friedrich L. Bauer
Mainz
Robert W. Bemer
New York
Theodor Erismann
Schaffhausen
Herman H. Goldstine
New York
Eiichi Goto
Tokyo
Motinori Goto
Tokyo
Walter Hoffmann
Zürich
Yasuo Komamiya
Tokyo
Noriyoshi Kuroyanagi
Tokyo
Tohru Motooka
Tokyo
Hiroji Nishino
Tokyo
Jan Oblonský
Praha
Willem L. van der Poel
Den Haag
Erwin Reifler
Seattle
Klaus Samelson
Mainz
Hans Konrad Schuff
Dortmund
Ambros P. Speiser
Zürich
Antonín Svoboda
Praha
Shigeru Takahashi
Tokyo
Hidetosi Takahasi
Tokyo
Rudolf Tarján
Budapest
Hideo Yamashita
Tokyo
Heinz Zemanek
Wien
Konrad Zuse
Bad Hersfeld

FRIEDR. VIEWEG & SOHN
BRAUNSCHWEIG

ROBERT W. BEMER
New York, USA

The Present Status, Achievement and Trends of Programming for Commercial Data Processing

With 4 Figures

Disposition

Summary. Programming for commercial problems requires all of the techniques necessary to scientific problems and a great many more. This paper documents some of the basic elements derived in the explosive development of programming techniques that has taken place in the last eight years, which is the short time that electronic data processing equipment has been applied in volume to commercial applications.

Programming costs are already a major portion of total data processing expenditures. This relative percentage may be expected to increase as new hardware advances come into production. It is therefore particularly important to assess the possibilities in reduction of programming costs through automatic techniques. Among these are machine-independent languages, program generators for special classes of recurring problems, program-hardware interactions, and total systems control programs.

There are several trends to be noted in programming methods. Among these are the automatic operating systems (with disappearance of the operator console), tabular languages, input-output control systems, the automatic production of automatic programming processors, remote operation of computers through communications links and corresponding service to small users, standardization of techniques and communication between different computers by common language. There is also an important trend to generalize programs and share them among many users of a particular class of machine through trade organizations.

Commercial programming has developed into a complex discipline of its own, with professional status. Technical education and publication therefore assumes an increasing importance.

Zusammenfassung. Die Programmierung von Problemen der kommerziellen Datenverarbeitung erfordert alle für das wissenschaftliche Rechnen notwendigen Programmiermethoden — und dazu noch viele weitere. Dieser Beitrag hält einige der grundsätzlichen Tendenzen fest, die sich im Laufe der stürmischen Entwicklung der Programmiertechnik etwa während der letzten acht Jahre herausgebildet haben, seit elektronische Datenverarbeitungsanlagen in größerem Umfange für kommerzielle Aufgaben eingesetzt werden.

Die Kosten für Programmierarbeiten machen heute schon einen beträchtlichen Teil der Gesamtkosten für die Datenverarbeitung aus, und es ist zu erwarten, daß der relative Anteil dieser Kosten mit fortschreitender Entwicklung der technischen Anlagen weiter wächst. Es ist deshalb besonders wichtig, die Möglichkeiten der Senkung der Programmierkosten durch automatische Programmiertechniken abzuschätzen. Hierzu gehören maschinenunabhängige Programmsprachen, Programmgeneratoren für spezielle Klassen von rekursiven Problemen, Wechselwirkungen zwischen Programm und technischer Anlage, sowie Programme zur Steuerung von Gesamtsystemen.

In der Entwicklung der Programmiermethoden sind verschiedene Tendenzen hervorzuheben. Dazu gehören automatische Bedienungssysteme (unter Weglassung des Bedienungspultes), tabellarische Programmsprachen, Ein-/Ausgabe-Steuerprogramme, automatische Herstellung von Programmübersetzern, Fernbedienung von Rechenanlagen mit Hilfe von Nachrichtenübertragungsgeräten und Fernbenutzung durch Kleinabnehmer, Standardisierung der Technik und des Verkehrs zwischen verschiedenen Rechenanlagen durch eine gemeinsame Programmsprache. Von Wichtigkeit sind auch die Bestrebungen, Programme zu verallgemeinern und einer größeren Anzahl von Benutzern einer bestimmten Typenklasse von Maschinen durch entsprechende Vertriebsorganisationen zur Verfügung zu stellen.

Die Programmierung auf dem Gebiet der kommerziellen Datenverarbeitung hat sich bereits in eine eigene komplexe Disziplin mit professionellem Status fortentwickelt. Dementsprechend kommt der technischen Ausbildung und den publizistischen Bemühungen auf diesem Gebiet eine wachsende Bedeutung zu.

Résumé. La programmation des problèmes de traitement des données commerciales exige toutes les méthodes qui sont également nécessaires pour poser numériquement les problèmes scientifiques, et encore quelques unes de plus. Le présent travail traite de quelques uns des éléments de base qui se sont développés pendant ces huit dernières années au cours de l'évolution rapide de la technique de la programmation et en particulier pendant la période relativement courte au cours de laquelle des installations de traitement électronique des données ont été mises en service sur une grande échelle pour résoudre des problèmes commerciaux.

Les frais de programmation représentent une partie importante des frais totaux dans le domaine du traitement des données. Ce pourcentage continuera apparemment à croître dès que de nouveaux progrès dans l'assemblage des machines seront appliqués à la production. C'est pourquoi l'évolution exacte des possibilités de réduction des frais de programmation par des procédés automatiques prend une importance particulière. On compte parmi ceux-ci les langages de programme indépendants de la machine, les programmes «produisant programmes» pour des classes spéciales de problèmes récurrents, les interactions entre le programme et la machine, et les programmes pour commander le système complètement.

Il faut remarquer diverses tendances d'évolution dans les méthodes de programmation; parmi celles-ci figurent, par exemple, les systèmes entièrement automatiques (dans lesquels on n'a plus besoin de panneau de commande), les langages de programme synoptiques, les programmes de commande d'entrée et de sortie, l'établissement automatique des traducteurs de programme, la télécommande des ensembles de calcul par l'intermédiaire des réseaux de télécommunication et service correspondant pour les petits utilisateurs, la normalisation des méthodes et l'échange d'information entre différents ensembles de calcul par l'introduction d'un langage de programme commun. Les efforts en vue de généraliser les programmes et de les mettre à la portée d'un nombre aussi grand que possible d'utilisateurs d'une classe déterminée de type de machine par des organisations correspondantes d'exploitation sont également très importants.

La programmation dans le domaine du traitement des données commerciales a déjà évolué vers une discipline propre complexe avec un statut professionnel. L'éducation technique et les efforts publicitaires prennent en conséquence une importance croissante.

1. The Environment

1.1 What Programming Is

The data processing or computing machine of today is provided with a repertoire of basic instructions or commands imbedded in the hardware. These instructions are the prime control for actual operation. However, such a set may be likened to the nervous system of man and its synaptic control of body movements and actions. Although these elements are mostly present at birth, education of the brain and related control centers is required for efficient action. The programming of a computer is equivalent to this education. The external command *"Write your name"* will, in a human, set in motion that stored program necessary to make the required movements automatically. The equivalent training could be given to the machine so that, upon receiving the same command in a natural language, circuitry would be actuated, storage would be searched, and the printer would be activated and print out *"I am a Siemens-Halske 2002"*.

In the quest for more speed and easier operation of electronic computers, it has been recognized that an equivalent amount of education is necessary to allow the human-machine-human input and output operation to keep reasonable pace with internal speeds. Programming systems in general correspond to a liberal education; programming for specific applications may be likened to on-the-job training in a specific field, utilizing the liberal education for easier assimilation. Comprehensive treatments of the principles of programming may be found in the literature [1 to 7].

1.2 The Need for Improved Programming Systems

During the first decade of mass usage of stored program computers, various devices have been developed to lessen the programming burden. These range from simple assembly programs through symbolic and mnemonic assembly programs, macro-instructions, interpreters, generators and compilers. All of these must be mentioned under the present state of the art because of the nonuniform advancement of segments of the programming population. Even today we find a large number of people, primarily in the area of commercial applications, still programming in actual machine language for one reason or another, particularly in the belief that this yields the ultimum efficiency in running time.

Such complacency is possible when there is but one machine operating eight hours or less per day, with only a few applications of a rather permanent and invariant nature. However, whenever the first machine of a type is utilized around the clock, or in multiple machine usage, or where the problems to be solved are many and varied, — then we find that communication with the computer must be accomplished in a higher level of language. Most experienced users of computing and data processing equipment are clamoring for advanced programming systems for these several reasons:

1.21 Complexity of Business Problems. The staff of programmers does not seem to diminish appreciably even after the first applications have been established and running. This is due in part to improvement and expansion which was not possible before and in part to continuing and awkward changes. The human clerk is well adapted to making changes and handling exceptions. For this reason, business users venturing into the computer field did not realize at first the volume and continuing nature of procedural change due to laws, competition, improved methods and management vagaries.

Scientific installations generally preceded commercial installations. The business user, looking at the large profits realized in scientific computing, was deluded into believing the same techniques were applicable to his operation. Unfortunately, commercial problems are in general at least ten times as complex as scientific problems. For one thing, there is no general language for business, as there is for mathematics. For another, many scientific problems are highly repetitive, which lends itself nicely to the looping facilities of stored programming. Thus a scientific program of 10,000 instructions is the exception, whereas a commercial program of 80,000 instructions is almost commonplace.

1.22 Ratio of Programming to Running Time. Elapsed time for initial programming and subsequent changes is presently too disproportionate to the actual running time on the machine. It is not unusual for an average application to require six months of programming and diagnostic correction to achieve a correctly running program. The computer must be prepared for every eventuality; the human clerk can stop and ask every time a strange condition arises. This reduces the time span in which competitive changes may be made effectively.

For the last five years, programming costs have been considered to be roughly equal to all other costs of operating data processing equipment, such as price or rental, installation, power, etc. Extrapolation of programming and engineering advances indicate that this percentage may well climb from 50 to 90 within the next five years. Machine speeds are now 100 times what they were five years ago. Programming cannot maintain this pace, nor will pouring armies of programmers into the gap help appreciably. The answer must lie in advanced programming systems which do a much larger proportion of the reasoning that humans now do.

1.23 Changing Technologies and New Machines. The evolution of new and improved hardware faces the user with additional problems. He finds himself with the opportunity to obtain a computer which will do the same job faster and cheaper than his present machine, yet his programming investment for the old machine must usually be considered a total loss. A possible amelioration is seen in adapting new technology to old logic, so that there is a family of machines with roughly the same basic machine language. This technique has found expression in simple transistorizing of machines that were formerly tube oriented, or in replacing electrostatic storage with magnetic core storage. This is self-defeating because the late members of such families cannot compete in power with new machines that have broken completely away and are balanced to the new technologies.

This dilemma has now occurred for one and possibly two generations of computers. The first time was not so difficult, for most users were not yet adjusted to the shifted emphasis required to convert applications from clerks or punched card equipment to the very different electronic data processing concepts. They were capable of making fresh starts with a new machine and their previous investments were not so great. As more investment accrued, yielding more experience, and the use of computers settled down to a predictable production pace, the costs of reprogramming a large volume of applications became staggering. Without the prospect of being able to program in a language independent of computer characteristics, the user must face an endless series of interruptions with changing machines.

1.24 New Methodology. Although many data processing installations simply converted existing methods and procedures to electronic equipment, additional efficiency and profit may be obtained by revising such procedures to correspond to the logic of the computer. This requires not only programming languages but additional ancillary languages suitable to the methods analyst, such as flowchart-ing, table structures, flow logic, etc., all oriented toward machine recognition and convenience.

1.25 Systems Concepts. An entire business system was formerly the servo-synthesis of the actions of a complex of humans with various responsibilities, however dimly understood. The hybrid mode of operating a data processing system under human supervision is not as satisfactory as having the supervision reside in program hierarchies within the machine. New languages are therefore needed to express executive concepts and translate them to machine action.

1.3 Opportunities in Commercial Applications

There is presently a great variety in the applications handled by data processing equipment, yet the total volume is nothing to what we may expect for the future, when such equipment is integrated with communications systems. Computers are among the most expensive devices manufactured. Unshared usage requires high volume. The shared usage of the future will reach a variety of low volume applications. Some of the present applications firmly established as profitable operations are:

Insurance, premiums and claims
General accounts, payable and receivable
Railway freight control
Petroleum reserves, product optimization
Tax gathering and verification, refunds
Inventory, shop scheduling, parts catalogs, spares
Shipping, transportation problem
Stock and bond trading, quotations
Livestock improvement
Personnel records, skills inventory
School curricula and grading
Real time process control
Banking, check clearing
Reservations and loading
Military defense systems
Optimum steel production
Merchandizing, order and reorder

Some applications which are just now coming into being are:

Mechanical language translation
Medical diagnosis, records [8]
Numerical machine tool control
Legal searching and correlation
Information retrieval, abstracts, library
Compressed communications
Business games, optimizing profit
Mechanical editing
Patent search

An exhaustive survey of governmental applications may be found in [9].

2. Programming Languages

2.1 Machine-oriented Languages

A general criterion for distinguishing a machine-oriented language is that programs written in this language will not run on another computer (not of the same generic family) unless under control of a simulation program which duplicates the characteristics of the original machine. Using as an example a machine with instructions consisting of one operator and one operand address, the various levels of convenience of representation might be:

00110001010101000110 (binary representation, no programming translation required)

3 1546 (decimal digits used for characters, input-output devices will accept and produce such characters)

H A623 (alphabetic characters as well in input-output devices)

RAD 20.66 (symbolic notation — *RAD* stands mnemonically for *R*eset *AD*d, 20.66 is an address number standing conveniently for the actual address eventually assigned by the assembly program)

RAD GROSSPAY (*GROSSPAY* serves the same purpose as 20.66 but is more convenient to the programmer for its mnemonic content)

Each refinement puts an additional burden upon the assembly program in the assignment and translation functions, removing this same burden from the programmer. This is justified by the assumption that the machine can perform these clerical functions with greater economy and fewer errors than the human. A further refinement is the addition of macro-instructions [10, 11], which are machine-like and compound. For example, any of the instructions

(a) *MOVE, TODAY, CURDT*

(b) *MOVE, TODAY, CURRENTDATE*

(c) *MOVE TODAY TO CURRENTDATE*

would generate for the 705 the instruction pair

RCV CURRENTDATE (*R*e*C*ei*V*e at the address for *CURRENTDATE*)

TMT TODAY (*T*rans*M*i*T* the contents of *TODAY*)

Such macro-instructions illustrate the correspondence between programming and hardware which sometimes leads to ambiguity and misunderstanding. For a machine with instructions consisting of one operator and two addresses, these would be simple instructions, not macro-instructions. Thus programming systems in effect redesign the hardware of a computer. Conversely it might be possible to construct a machine which accepts, as instructions built into hardware, an algebraic language such as ALGOL.

The common practice of limiting the number of characters in symbols employed by the programmer is illustrated by (a). This is done for two reasons. First, in a machine which moves information by words (groups of bits addressable uniquely) it is uneconomical to use more than one word to represent a symbol. In most present machines 6 bits are used internally to represent a letter or digit, thus a machine with 24-bit words handles no more than four character symbols conveniently. Second, since these names must be carried through the translation process, they become a more expensive burden as an increasing number of manipulations are made. A better practice is to assign a working number (or address number) to each symbol, in a table of double reference, more particularly because the set of meaningful alphabetic symbols is very sparse; i. e., *GXPQ* carries no more mnemonic content to the programmer than 63987. (Cf. [12].) The number of characters in each symbol may then be unlimited. COBOL [13], for example, restricts the length of symbols to 30 characters for practical convenience.

More meaningful symbols may be used as separators or punctuators for clarity to the programmer as it is shown by (c). The translating program can be made to recognize this function and even to accept extraneous *noise* words. Thus the so-called *"English language"* of the FLOW-MATIC Programming System [14] for UNIVAC I is basically a method of employing three-address macro-instructions with freer form. Thus

ADD A, B, C is equivalent to *ADD A TO B GIVING C*

The difference lies in the fact that commas, as separators, do not order the process either mnemonically or logically. The programmer must know the function of each of the operands as written in the macro-instruction. The alternate form of FLOW-MATIC is still restrictive logically. A rather rigid form is still required. For example, it is not permissible to vary the previous instruction to

TO B ADD A GIVING C

Although proper English, such logical ability is not inherent in the formation rules of the restricted and artificial programming language and is therefore not reflected in the translators. The macro-instruction *ADD A, B, C* is even more ambiguous, however, since only by definition (which the programmer must remember) is it known whether it means

$A = B + C, \quad A + B = C,$ or even, although unlikely $B = A + C$

The same operation may be expressed in a number of alternate forms, any or all of which may be acceptable, *provided* that the recognizing and translating mechanisms are incorporated in the processor.

$C = A + B$
$A + B = C$
SET $C = A + B$
REPLACE C SUM A AND B
REPLACE C BY THE SUM OF A AND B, etc.

BY, *THE* and *OF* are examples of extraneous noise words added for clarity. They are ignored by the translator.

Macro-instructions are normally of two types, *library* or *programmer*. Library macro-instructions are those useful to a general class of problems and are thus automatically available in the processor, which recognizes them by table-scanning to be different from the one-for-one representation of a single machine instruction. Quite often these may be of a complex generative nature (usually formed under control of a matrix of alternates) which alters or eliminates redundant instructions for efficiency.

Programmer macros, in contrast, are those specific to this particular problem. Usually some sequence of instructions is virtually repeated in many places with slight or no variations. The programmer, recognizing this, defines the general case by example (which is assimilated by the processor upon recognition as such) and thereafter saves much copying and possible error by its use. Processors commonly recognize this type of instruction by finding the term *MACRO* instead of the normal operation code. The following Example 1 shows how a programmer macro may be defined and later used.

Example 1

Name	Operator	Operand Field	
SUMPROD	*MACRO*	*A, B, C*	
	ADD	*A, B, C*	$(A+B \rightarrow C)$
	MPY	*C, B, C*	$((A+B) \cdot B \rightarrow C)$
	MPY	*C, A, C*	$((A+B) \cdot B \cdot A \rightarrow C)$
	.		
	.		
	.		
	SUMPROD	*X, G, Y*	

The last instruction thus computes $Y = X \cdot G \cdot (X + G)$. This device may be used very effectively when the number of instructions created is large and when other macros of the same type are used recursively in the definition of a still larger macro. This will be recognized as the genesis of the **procedure** *statement* in ALGOL.

The Example 1 also illustrates the artificial time sequence of input to a translator. Normally the sequence of entries is mapped into the sequence of instructions executed, whether in contiguous sequence or chained (as in drum storage machines). Here, however, the instructions of the example are never executed in their own right; they are merely dummies. (Cf. Figure 4 for automatic reorganization of the program through addition, deletion and replacement.) Literals and operators may also be varied within macro-instructions. (Cf. [15].)

Macro-instructions are essentially open subroutines and are placed in the main line of the program, whereas closed subroutines require transfer instructions and are set up by a calling sequence or linkage. Each time the macro is used, a copy of the instructions generated is placed in direct line in the program. It is not to be supposed that macro-instructions yield but a few machine instructions while subroutines have many. The output of macro-instructions may also be formed as a subroutine. The proper way to build a macro-instruction generator is to equip it with the facility for self-determination of whether to insert in-line or as a closed subroutine. The pròcessor may contain a program section which weighs available storage space against execution time and the number of times used, for the closed subroutine consumes more running time by virtue of the extra calling sequence required to set it up to operate.

The expansion of a macro-instruction through Autocoder language into machine language for the IBM 705 is illustrated in the following Example 2 which is an excerpt from a two-tape merge problem taken from [16]. The sense of the macro-instruction is to turn off the error indication triggers whenever a signal is received from any of them.

Example 2

Macro-instruction written in Autocoder language:

```
            DOA   XOFF
```

Basic Autocoder instructions incorporated in program:

```
⌑⌑000005  LOD    14  ⌑⌑000005
          TRA        XOFF
XOFF      UNL    14  XOFF3        TYPEWRITER INDICATOR OFF
                                  SUBROUTINE
          LOD    14  #0010#
          ADM    14  XOFF3        TO RETURN ADDRESS
          SEL        901
          TRS        XOFF2        TURN OFF 0901
XOFF2     SEL        902
          TRS        XOFF3        TURN OFF 0902
XOFF3     TR                      RETURN TO MAIN PROGRAM
```

This produces basic machine instructions as:

```
      Location    Contents
      0524        80EK4
      0529        11609

      1609        71FM4
      1614        81GN2
      1619        61FM4
      1624        20901
      1629        O1634
      1634        20902
      1639        O1644
      1644        10534

      1752        0010
```

2.2 Procedure-oriented Languages

The distinguishing feature of a procedure-oriented language is that its syntax is not necessarily related to that of the machine language for a particular machine. There are many similarities; the need still exists to specify the procedure or algorithm for problem solution in terms of time-dependent steps, both logical and arithmetical. In general a procedure-oriented language depends upon imperative statements. However, machine languages are introspective; procedure-oriented languages are normally not introspective. That is, the specific characters of the imperative statement may not be operated upon by actions initiated by another imperative statement. The introspective nature of machine language may be illustrated by the instruction group of the following Example 3.

Example 3

Address	Operation	Operand
0001	*SET*	0046
0002	*RESET ADD*	0001
0003	*ADD*	0020 (The contents of 0020 are 0027)
0004	*STORE*	0001
0005	*TRANSFER*	0001

The instruction in 0001 will now be *SET* 0073 when obeyed. A corresponding example for a procedure-oriented language may be constructed by mapping

Address: Operation, Operand *into* Label: Statement

Introspection is now illustrated by the following statement group:

SUBSTITUTE: REPLACE LAST DATE BY CURRENT DATE PLUS 3.
: CHANGE OBJECT OF (SUBSTITUTE) TO NEXT TO LAST DATE.
: GO TO SUBSTITUTE.

It will be seen that scanning and logical difficulties are magnified greatly in the latter set. Thus there is little introspection in present languages of this type.

A good test for validating a procedure-oriented language is to determine whether a human could follow the procedure manually with limited knowledge of data-processing equipment. This is one of the two reasons for the design of such languages – broader understanding with less experience. The second reason is that casting procedures in this form leads to limited independence of machine type, so that large sections of program will not have to be rewritten for a second machine.

Some examples of such languages are FLOW-MATIC [14], Commercial Translator [17], FACT [18], and (on the scientific side) ALGOL[1]. The first great advance made in such languages was the separation of the program into two parts, procedure and data description. In using machine languages, the characteristics of the data (operands) are implicit in virtually every instruction. Since the statements of a procedure-oriented language must go through a computer translation process, it is economical to give the full characteristics of the data only

[1] Reference is made to the contribution by F. L. Bauer and K. Samelson, in this volume pp. 227–268.

once and let the machine program produced be the intersection of the procedure and the data description. This is a considerably more complicated process and costs more in machine translation time, but it has been found that the advantages in reduction of human cost and error, plus the multimachine freedom from reprogramming, more than compensate.

A further advance was made in Commercial Translator, with the breaking out of a third section on environment. Here the machine features (how many tapes, what storage size, etc.) need to be specified only once, and the resultant program is the intersection of all three inputs. Commercial Translator also introduced the concept of logical multipliers, assigning arithmetic values of **1** to truth and **0** to falsity. Thus the statement:

> *DISTANCE* = 500 − 60 * *TIRED* (the asterisk signifies multiplication, either logical or arithmetic)

If *TIRED*, as a characteristic of some variable, is *true*, then

> *DISTANCE* = 500 − 60 * (1) = 440

If *TIRED* is not true, then the modifying term disappears and *DISTANCE* = 500.

Another vital concept in such languages is that of logical brackets to delimit the two resultants of a conditional statement. Consider the statement

IF A=C THEN IF A>B THEN DO K ELSE DO L ELSE IF B>A THEN DO M ELSE DO N.

This statement represents the flowchart of Figure 1.

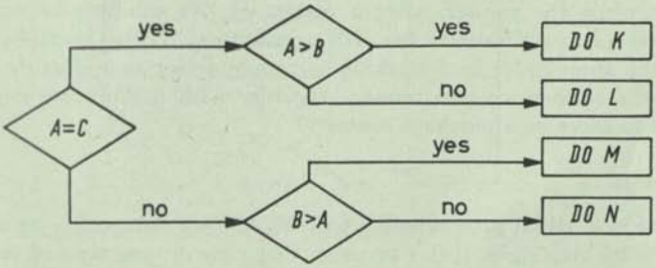


Fig. 1. Flowchart representing the statement
IF A = C THEN IF A > B THEN DO K ELSE DO L ELSE IF B > A THEN DO M ELSE DO N

Obviously *THEN* and *ELSE* are logical brackets [19] which could be represented by single symbols for easier understanding, thus:

> *IF A=C ⟨IF A>B ⟨K⟩ ⟨L⟩⟩ ⟨IF B>A ⟨M⟩ ⟨N⟩⟩*

It is not our intention to introduce herewith such a bracket notation; it is only to clarify the principle. The ALGOL language, for instance, in the author's opinion is not quite as recursive as it possibly could be, and therefore not as convenient in comparison to the less restrictive syntactical mechanism present in the statement of Fig. 1. The awkwardness in ALGOL, due to the definition of the **if**-statement which makes mandatory the provision of the statement parentheses **begin** and **end**, increases with more complex flowcharts. We must be aware that business problems often structure themselves in much more complicated logical patterns than do scientific problems.

2.3 Problem or Goal-oriented Languages

A problem-oriented language is distinguished by the lack of a stated procedure for solution. The responsibility for creating the object program as a procedure in machine language belongs to the processor, which presumably allows for all known variations in this type of problem. The method of problem solution is implicit in the intelligence of the processor, which is variable and may be augmented. Two classes of such languages are presently in volume use, the ordering (or sort) generators and the report generators. (Reference is also made to Section 4.)

Obviously such processors are suitable only for highly repetitive types of work which will justify the expenditure of creating the language and programming system. Processors for problem-oriented languages are special purpose as contrasted to the general purpose nature of machine- or procedure-oriented languages. There is a direct analogy (as there usually is between programming and hardware) with the building of special purpose computers for recurring applications, since they can be more efficient than a general purpose machine, although limited.

Other specialized languages have been created, and fall in this class. Examples are the languages for automatic control of machine tools [20], operating systems for computers [21 to 25], tabular languages [26 to 28], and even languages for design problems. This latter class is exemplified by a program especially created for the design of electrical transformers. The transformer manufacturer allows the prospective customer to fill in a form (surely a type of language!) with his own specifications and requirements, see them entered into a computer and watch while the printer, after a matter of five minutes or so, writes a complete set of specifications for that tranformer. These specifications also comprise the shop order and working information for manufacture, a bill of materials, the sales price and terms, — together with a duplicate copy with a dotted line to serve as a purchase contract!

2.4 Simulators

A simulator is a useful programming tool which does not qualify as a language processor in its own right. It is a program that runs on one type of computer to simulate the action of another type under control of a program written in the language of the second computer. Thus a program that runs on a *Mercury* Computer could also be run on a UNIVAC 1103 under control of an interpretive simulator, if one happened to have been written. Simulators are useful under the following circumstances:

(a) During transition to a new machine. If the new machine produces *W* times as much work per cost unit as the old one, and if the simulator runs no more than *W* times as slow as a direct program, the simulator will be useful to run until the programs are rewritten and checked out for the new machine. Not only is the cost of producing the simulator neglected here, but it may be advantageous to incur time losses to effect the transition.

(b) To check programming systems written for a machine not yet manufactured, so that the systems will be available with machine delivery.

(c) To check production programs on an existing machine before releasing and displacing it by the new machine.

(d) In mixed machine installations to compensate for unbalanced work loads. If machine *A* is overloaded and *B* is available, simulate some of the *A* programs on *B*.

A sometimes cheaper means of achieving (b) is to write the translation processor for the new machine directly in its own language and then rewrite that same translation process as an application problem on another machine for which a comparable program exists. The processor is then assembled on the old machine, producing a machine language program for the new machine. From this point on, both the old and new programs are useful, depending upon which machine is available [29].

3. Elements of Programming Systems

There have been three basic stages in the solution of problems by data processing equipment. The first is illustrated by Method I of Figure 2. The entire process of planning and coding the solution is done without machine aid. Primitive programming systems are exemplified by Method II of Figure 3. This is basically the assembly method, but with some modification an equivalent interpretive system could be constructed.

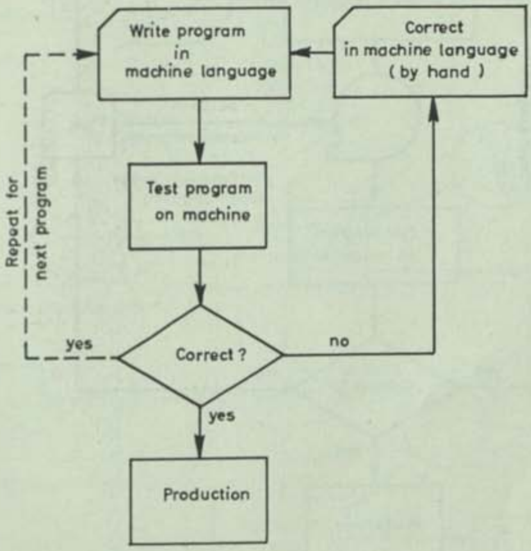


Fig. 2. Solution of problems by data processing equipment
Method I: Planning and coding without machine aid

The third stage in development of programming systems had its origins in the supervisory system concept. With the flux of new ideas in the operation of stored program computers, it is difficult to get general agreement on just what elements should comprise an operational system. A composite, with much latitude of definition, might consist of:

Executive Control or Supervisor
Translator – Assembly Language to Machine Language
Translator – Procedure Language to Assembly Language
Diagnostic Section
Input-output Control System (IOCS)
Macro-instruction and Subroutine Library
Application Library
Ordering Generator
Report and File Maintenance Generator } (treated in Section 4)

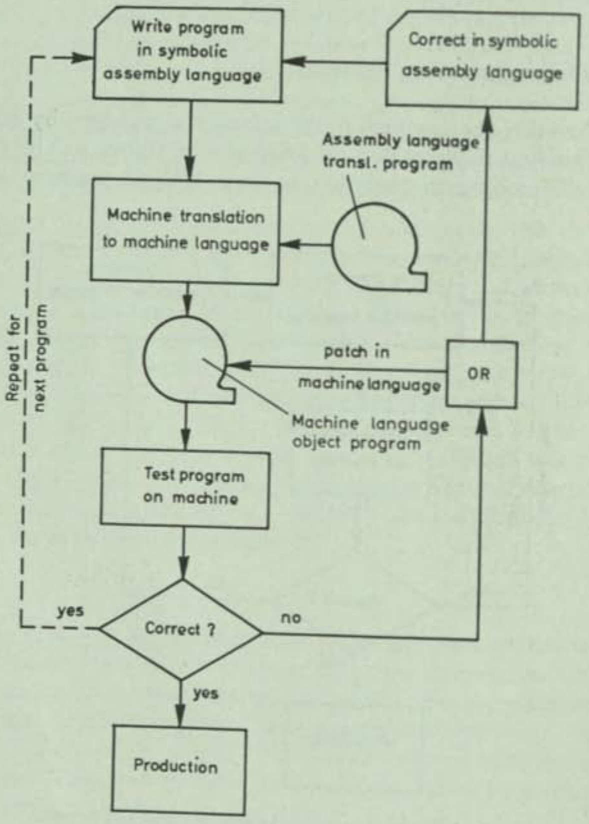

Write program in symbolic assembly language
Correct in symbolic assembly language
Assembly language transl. program
Machine translation to machine language
Repeat for next program
patch in machine language
OR
Machine language object program
Test program on machine
yes
Correct ?
no
yes
Production

Fig. 3. Solution of problems by data processing equipment
Method II: Primitive programming systems

All of these are best interconnected by various control elements (programmed) and the entire system is called a *processor*. Applications programs, which obtain specific answers, are normally called *source* programs. These are converted by the processor to running, or *object*, programs which are then executed, still under the control of the overall processor. It is realized that many contemporary data processing installations provide only manual linkages to interconnect these various phases. Method III of Figure 4 represents the computer in full control of its own operation, subject to manual override.

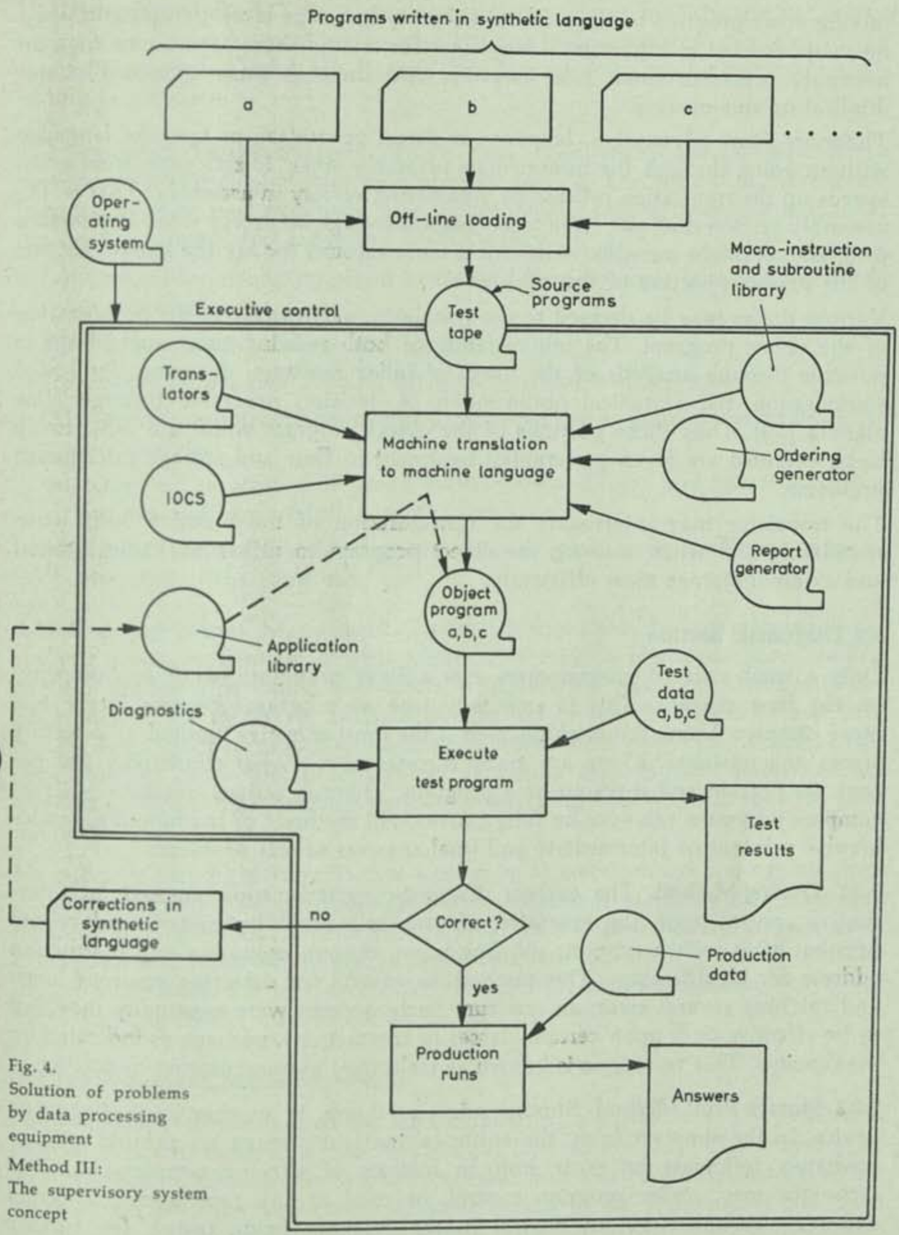


Fig. 4.
Solution of problems by data processing equipment
Method III:
The supervisory system concept

3.1 Translators

There is considerable variation in the capabilities and duties of that element which translates from the source to the object language. The translation procedure might utilize a two-stage process from synthetic language through an intermediate assembly language to machine language. This allows for intermixing other program sections at the assembly language level, particularly when no methods exist in higher level languages for stating these procedures. Such an assembly processor must exist anyway, and there is some economy in not duplicating this process.

There are some advantages, however, in direct generation of machine language without going through the intermediate assembly stage. Direct generation often speeds up the translation process by eliminating variety in assembly. That is, the assembly section does not have to be general enough to accept what any human programmer might possibly write. All it must account for are the known actions of the previous section of the translator.

Various duties may be charged to the translator, particularly in the optimization of the object program. The minimization of both running time and storage is possible through analysis of the usage of index registers, detecting duplicated computation and statistical optimization of decision processes through flow algebra [30]. Thus those portions of the object program which are most likely to be executed are given preferential treatment in flow and storage interchange problems.

The translator may interrogate the configuration of the machine both when translating and when running the object program to utilize available internal and external storage most efficiently.

3.2 Diagnostic Section

Only a small core of programmers ever achieve programs which run correctly on the first attempt. This is especially true as programs become larger and more complex. Using the machine itself is the most effective method of detecting errors and mistakes. There are many theories about what constitutes 100 per cent verification and checkout of a program. There is serious question whether complex programs can ever be fully proven. All methods of machine diagnostics involve printout of intermediate and final answers to test problems.

3.21 Tracing Method. The earliest diagnostic systems superimposed an interpretive control upon the execution of each individual instruction and caused printout after each execution, showing input, output, operation and instruction address for identification. This method was useful for detecting spurious loops and catching several errors in one run. Such systems were eventually modified to be effective only upon certain classes of instructions, perhaps as indicated by breakpoints. This technique is known as (selective) automonitoring or tracing.

3.22 Storage Print Method. Storage print, or dump, is another basic diagnostic device. In the simplest form, the entire contents of storage are printed out immediately following an error stop, in formats of varying complexities. This technique may, under program control, be used at any time during program execution and then return control to the program being tested, for further execution. In the more advanced forms, the original contents of storage are retained on tape, compared against the contents at stop time, and only the changed portions of storage are printed. This avoids much tedious human search and isolates difficulties quicker. Usually some form of conversion is applied to storage before printing, particularly in a binary machine, whose dump would be rather unintelligible. Sometimes areas of storage known to contain instructions are transformed to assembly language form.

3.23 Snapshot Method. This is similar to tracing except that the method is not interpretive. Instead, actual and precise printing instructions are compiled in the symbolic program. They are flagged for easy and automatic removal when testing is complete.

3.24 Automatic Testing Systems. There have been many special supervisory systems written exclusively for bulk program testing. Test time is at a premium, particularly with new machines, and unmechanized testing with the human at the console is much too expensive. Such systems:

(a) Test, in series, the programs of many different people who are preferably not present.

(b) Reduce manual and console operations.

(c) Ensure proper tape loading for each program without time lag.

(d) Generate various classes of test data to exercise as many program branches as possible.

(e) Keep full records of all stops, addresses, conditions and operator actions for easier diagnosis following the run.

(f) Feed in corrected data after an error so other error conditions may be detected in the same run.

3.25 Running Checks. Many checks and verifications may be incorporated in the running program. Records written erroneously may be identified upon reading through the use of so-called *hash-totals,* which are not totals associated with the program but are rather an artificial summation of the characters or bits in a record, or group of records. Records may be automatically corrected if augmented by Hamming check bits or characters. Normally records with flaws are not allowed to stop the processing; they are written out on exception files for later handling.

Checkpoints may be incorporated in a program at a convenient break point (i. e., integral processes are fully completed). Zero balance or matching data tests may be made here. If errors are detected, the program is returned to the last succesful checkpoint and restarted; if everything is checked, this point is installed in the proper address as the last successful checkpoint. A comprehensive survey of other auditing checks may be found in [31].

3.3 Input-output Control Systems

Approximately 40 per cent of the total number of instructions in a typical commercial program will pertain to input-output operations [32]. This includes all data movement through the central processor and related housekeeping which must accompany this movement. A major saving to the programmer has been the development of specialized input-output control systems (as integral portions of the entire operating system) which can do the following functions automatically:

(a) Match tape labels to unit numbers, verifying correct mounting of data, system and program.

(b) Detect tape type or density, when more than one recording density is available.

(c) Prevent erroneous writing on tapes which contain permanent or semi-permanent records.

(d) Flip-flop tape units on symmetrical jobs, such as ordering.

(e) Maintain an automatic count of records entering and put out.

(f) Alternate input-output area usage in internal storage.

(g) Relocate programs in storage for multiple operation.

(h) Optimize read-compute-write overlap.

(i) Automatic unblocking of records for reading and blocking for writing.

(j) Housekeeping associated with tape files, such as rewind, error correction.

(k) End of reel operations in multi-reel files.

(l) End of job functions, such as logging time, notifying next user, upspacing printed records, labeling tape files created, etc.

(m) Automatic insertion of checkpoint and restart procedures pertaining to input-output operations.

Such an input-output control system package is essential to the operation of language elements such as *GET* record, *PUT* record, *OPEN* file and *CLOSE* file which appear in Commercial Translator, COBOL, etc.

3.4 Application Library

The use of the application library requires full system control. In the translation from source to object program, many items of information are gained at the expense of computational time and many specific decisions are made. Running problems according to Method II (cf. Fig. 3), a complete reprocessing must occur each time the source program is corrected. Virtually the same information must be regenerated with a corresponding time expenditure. The application library consists of current source programs maintained on magnetic tape or other external storage, not only in their original (or source) form but also in the latest object form, together with the storage assignment and other tables and information developed during the latest translation.

As changes are made, they are identified by the name of the application. The executive routine searches the library for that program, copying the library to another tape until it comes to the desired program. This program is now brought into internal storage, corrected in source form and translated to a corrected object form. All updated information on this program is now copied to the new tape and all succeeding programs are copied from the old to the new tape simultaneously with the test execution of the corrected program. Thus the object program is corrected from information delivered in source form without operator intervention in only a fraction of the time a complete reprocessing would take.

3.5 Macro-instruction and Subroutine Library

In the most flexible state, a program may consist of a mixture of machine-independent statements, algebraic equations, macro-instructions, symbolic machine instructions and actual machine instructions. The scan in the translator is responsible for separating these into classes during processing, retaining an exact indication of the ordering which indicates flow. Macro-instructions are identified by the machine language-like form and by the fact that pseudo-operators do not exist in the table of machine operation mnemonics. A table of macro-instruction operators for which generators exist in the library is maintained in storage during the scan.

After initial scanning, macro-instruction calls are grouped and reordered to the order in which the generators appear on the library tape. The library tape is then passed against this list and all generators called for are extracted. After generation and establishment of symbolic addresses, the generated groups of instructions are ordered again to the original sequence in which they were called for by the program and merged with all other instructions to be generated or assembled. Unless this method were followed, there would be a series of tape searches and rewinds for each macro. Also, duplicates do not require additional searches in the reordering method.

Closed subroutines are handled in much the same way, except that calling sequences and return linkages are written by a single standard generator.

4. Retrieval of Information and Updating of Files

Files of data are maintained for specific purposes including display of individual data, search by classes, listing, access by other programs on demand, etc. There are several classes of generalized programs particularly concerned with this process. They are:

(a) File maintenance and updating generators.

(b) Report generators.

(c) Ordering and merging generators.

The first two apply to any type of file, the last applies only to files which are effectively linear, such as magnetic tape, and not to random access files.

A file is a collection of data (on some storage medium) which displays groups of similar properties. The individual elements of files are called records. A record contains both the actual data needed and other data which serve to identify that record from all other records. This identification is known variously as the key, control field, label, name, identification number, etc. Such files are either sequenced or randomly ordered, according to the storage medium upon which they exist. Particular records are found correspondingly by either examining keys through a prescribed search pattern or by transforming the key to a secondary locator.

A deck of punched cards, a magnetic tape and a perforated paper tape are all examples of sequenced files. They may be ordered by time sequence or key, i. e., it may be desired to find the 18th record in a file or that record containing the data on *Smith, H. J.,* for instance. In the latter case, a multiplicity of searches may make it profitable to order the file alphabetically upon the key, rather than scan the entire file each time (in random or linear order) until the key is found to match the given key. This characteristic has accounted for perhaps 30 per cent of the operating time in today's commercial data processing. This figure is not appreciably affected by random access files, which are a minority. Clearly this has been an area for profitable improvements in the reduction of programming and operating time. Since the problem is algorithmic, all ordering procedures are similar in principle and vary mostly in details. Such is the origin of the complicated and highly specialized ordering (miscalled "sorting") generators of today.

4.1 Ordering Generators

Because of the relatively greater cost and access time required to retrieve data randomly from files, the ordered file is still more economical for a large share of data processing needs. Ordering is a two-stage process. The user provides the generator with specific choices of, and statements about, the required input parameters. The processor digests this information and produces a specialized running program for these specific conditions. The actual program produced is the result of modifying skeletal sections of program with computational results, and is then utilized to order the files. Except for certain special and largely invariant conditions, these machine-generated programs are cheaper to produce and more efficient to operate than those created by the average programmer-user. This is because they are the product of specialists that can consider a larger spectrum of applications, and because of certain invariant principles. An advanced ordering generator might require the user to specify:

(a) The file size (number of records) and organization (whether on a single or multiple tapes, and how these are labeled for identification).

(b) The machine model and particular configuration of components available for this job.

(c) The number of magnetic tape units available for either mounting the files or intermediate transfer of information, such as record rearrangement.

(d) A choice between physical rearrangement of records in the intermediate steps or rearrangement of tags which identify or symbolize the particular records, reserving the physical rearrangement of the entire file until the sequence is fully determinable.

(e) The amount of internal storage available for use by each phase, or stage, of the process.

(f) The length of the records, whether fixed or variable length, and (if variable) how the length may be determined.

(g) The length of the key and its placement (or the placement of its components) in the record.

(h) The ranking or marshalling (term used in England) order of the characters from which the key may be formed.

(i) Existing partial ordering or bias in the data to be ordered, if any.

Many considerations are removed from the concern of the user by being incorporated in the intelligence of the generator. Among these are:

(a) Overlap of read-compute-write operations where feasible.

(b) Choice of ordering method (digit, merge, distribution, internal ordering, sifting, etc.) or a combination of several of these techniques as required to best utilize the machine in the various stages of the process (unless specifically countermanded by the operator). (Cf. [33].)

(c) Internal or input-output buffering.

(d) Blocking (grouping) of records for faster transfer within storage or between media.

(e) Automatic padding, or filling, of incomplete blocks or groups of blocks so that the file is modular for regularized processing. Automatic removal of padding on completion.

(f) Automatic replacement of keys by working keys whenever the internal character code of the machine does not have binary correspondence to the desired ranking order. Automatic replacement of original keys upon completion of the process.

(g) Collection and transformation of all elements of a key into a contiguous unit for convenience of comparison, with later dispersal to original format upon completion of the process.

(h) Calculation of estimated running time to completion, and advising the operator.

(i) Balancing the process as a function of the ratio of average computation time to tape read-write time (function of tape passing speed and bit density).

(j) Assignment of actual addresses to instructions, input-output units, etc., with provision for symmetric exchange of functions during the process.

(k) Automatic incorporation of rerun and checkpoint routines, for use in case of machine failure or detection of bad data. Provision for interruption at controlled points for jobs with higher priorities; thus ordering may be resumed at a later time without loss. This is vital because many files are so large that it might take from 1 to 20 hours of continuous time on the fastest machines.

An excellent description of some of these routines, with application to many machines other than those manufactured by IBM, may be found in [34].

4.2 Report Generators

It may well be that someday the control and management of business will reside within the computer program. In the meanwhile, decisions are still made by humans on the basis of condensed and categorized information prepared by either other humans or data processing equipment. The normal form of such a summary is the printed report. Here again the process of preparing reports is algorithmic and is thus suitable to action by a generator program.

The report generators create running programs which will abstract information from one or more files as needed to construct a specific report, rearranging and editing this information as required by the format of the report. (Cf. [35 to 37].) The user normally supplies the generator with the following information:

(a) The characteristics and format of the records in the files to be used.

(b) A pictorial layout or description of the report format, indicating spacing within the line and other editing conditions.

(c) Special instructions on printing or indicating various levels of totals, etc.

(d) Which different reports are to be printed on this one run, or passage of input data.

(e) Order of rearrangement of data in case the file is in a different order.

(f) Conditional printing desired (group indication, where information is repetitive).

(g) Rules for insertions and deletions in the input file, if file maintenance is incorporated in the same run.

Typically, much of the input information supplied is identical to that required for ordering generators, thus the two processes are often combined. Most report generators contain, in varying degree, the ability to be linked with other programs, to perform simple arithmetic necessary to production of totals, to perform file maintenance and updating, and to be modified at programmer discretion with inserts of assembly language subroutines.

4.3 Random Access to Files

Files may be searched in three basic ways:

(a) The scan, or random search, method to find the record with a matching key. This is prohibitive in cost except for very small files.

(b) The search of a file ordered on some function of the keys, such as alphabetic or numeric sequence properties. There is expense in initial ordering time and in addition to or deletion from the file. However, it is well suited to linear files and batch processing. The search method is most commonly binary or in a Fibonaccian sequence. The binary search is most prevalent and consists of successive partitioning in halves, selecting the half in which the required record must exist by checking the limiting keys against the desired key, and successive reduction until only the desired record remains.

(c) The search of a file located in storage by some algorithmic function of the keys. The key for which the record must be found is then subjected to the same algorithmic function to yield the address where the record is probably located. The only reason it may not be there is because of possible duplications in the values yielded by the algorithm over the entire set of keys. The better method of this type is known as *chaining*. (Cf. [38 to 40].) Although an inherently simple process, it is often misunderstood because of confusion about the handling of duplicated addresses.

Assuming the file is loaded, the chaining method requires that the key be converted by the algorithm to a tentative address. The key is then compared to the key existing in this address. If they match, the further contents of that address are those desired. If they do not match, a further address is also contained within the location specified by the tentative address. The key in that address (the chain address) should then be matched against the search key. The process is recursive until the proper key and address are found.

Let us take a simple example to show the loading of N records into P positions. If $N=P$, the file is 100 per cent packed or loaded. The following Example 4 involves the data for 13 names, or keys. The algorithm chosen from the myriad possible is:

$$\text{Tentative address} = \sum \text{(Letter position in alphabet)} \text{ modulo } P$$

This evaluation is made in Example 4. The duplications which occur will be handled by chaining. Alternate algorithms might be found which minimize such duplications, but in general it is not worthwhile to waste time in searching for a slightly better algorithm. $P=N=13$ in the example. The file is now loaded initially in the order in which the names appear.

Example 4

Name	Computation	Σ	Tentative Address = Σ mod 13
John	10 +15 + 8 +14 =	47	8
Fritz	6 +18 + 9 +20 +26 =	79	1
Klaus	11 +12 + 1 +21 +19 =	64	12
Julien	10 +21 +12 + 9 + 5 +14 =	71	6
Grace	7 +18 + 1 + 3 + 5 =	34	8
Walter	23 + 1 +12 +20 + 5 +18 =	79	1
Roy	18 +15 +25 =	58	6
Stan	19 +20 + 1 +14 =	54	2
Alan	1 +12 + 1 +14 =	28	2
Heinz	8 + 5 + 9 +14 +26 =	62	10
Rene	18 + 5 +14 + 5 =	42	3
Bob	2 +15 + 2 =	19	6
Peter	16 + 5 +20 + 5 =	64	12

Example 5

Address	Method I				Method II			
	Chain	Name	Data	Seeks	Chain	Name	Data	Seeks
0		*Grace*		2		*Grace*		2
1	2	*Fritz*		1	4	*Fritz*		1
2	4	*Walter*		2	7	*Stan*		1
3	7	*Roy*		2		*Rene*		1
4	5	*Stan*		2		*Walter*		2
5		*Alan*		2	9	*Roy*		2
6	3	*Julien*		1	5	*Julien*		1
7	9	*Rene*		2		*Alan*		2
8	0	*John*		1	0	*John*		1
9		*Bob*		4		*Bob*		3
10		*Heinz*		1		*Heinz*		1
11		*Peter*		2		*Peter*		2
12	11	*Klaus*		1	11	*Klaus*		1
	Average Seeks=1.77				Average Seeks=1.54			

Method I of Example 5 shows the result of loading when duplications are assigned to the first available open position. Thus *Grace,* the first conflict, is assigned to zero position. Method II shows the result if a different rule is used, holding all duplicates aside until the list has been gone through once, then loading into the available vacant positions. The number of seeks required to find each item at random has been tabulated. Note the improvement due to Method II. The scan, or random search, method of random loading would average seven seeks per record.

$$\left(\sum_{1}^{13} N\right) \div N$$

It has been shown [41] that the average number of seeks with random frequency distribution will be 1.5.

The average seek number can be improved by a number of techniques. Obviously $P > N$ will do so, but the advantage has been found in actual practice to be insufficient to use anything other than 100 per cent loading. Advantage may be taken of natural characteristics of data. Use of the method on the 305 RAMAC has yielded average seeks of 1.2 for fully loaded files. This indicates [38] that the average commercial problem will interrogate 20 per cent of the file 80 per cent of the time. The average may become as low as 1.1 for loading on a fully statistical frequency basis. This method was used to convert the English vocabulary to numbers [42]. It was found that the natural frequency of English usage yielded an average seek of 1.14.

The chaining technique is very helpful in translating programs to convert the names of variables to working address numbers for faster processing (cf. [12]).

5. Factors Influencing the Level of Programming

5.1 Logistics of Machine Configuration

The largest single factor affecting the advancement of the programming art is the logistic structure of computing machinery. Data processing equipment consists of much more than a central processing unit with arithmetic and logical decision capabilities. The availability of various hierarchies of storage facilities, various input and output devices (both on- and off-line), printing devices and character sets all have a profound effect upon the improvement of techniques.

5.11 Character Sets. For example, the lack of other than numeric input-output facilities in most Russian computers has seriously slowed development of synthetic languages for communication with the machine. Even in scientific computations, where Russian algebraic compilers have shown promise in the area of efficient optimization [43, 44], the programmer is unable to refer to a storage location by the name of its contents and have the computer operate directly upon this information, automatically assigning an actual location in storage. Since alphabetic characters are not available, a slow and inefficient process of transcription is necessary. This must be done by hand before the program may be entered into the machine, no matter how advanced the system is on paper. Thus an apparently trivial feature heavily affects operating philosophy.

The stored program machine is a general purpose device. We have realized for years that one of the problems it may be given is the automatic translation from the language of the programmer to its own but this is not easily accomplished when the basic elements of the language, the characters, are not common to both languages.

Alphabetic and other special characters are more available on computers in USA, but programming languages may still be found using phrases such as "*if greater than or equal to*". If a single symbol were available to the user to represent this phrase and others, processing time could be greatly reduced. As it is such phrases must be written out in their entirety by the programmer in longhand with the attendant possibilities of error and faulty decoding. A sample statement describing income tax deductions in Commercial Translator required 700 bits of information at 6 bits per character (a maximum of 64 symbols available). If three new characters could be added to the set, the total number of bits required would be reduced to 500 even if all characters had to be represented by 8 bits rather than 6. (Cf. [45].)

Interest in a larger set of characters has been increasing, largely influenced by the ALGOL language which presently has 110 characters for use in the reference language. This may improve general communication with the machine in all areas, and may prove to open new applications in computer-controlled typography. Some new machines, particularly the IBM 7030, are designed to handle larger character sets [46]. The *Bendix* input-output typewriter handles all the characters of ALGOL in an 8-bit form. *Ferranti* and *Bulmers (Friden)* in England have made provision for 7-bit sets for input and output.

5.12 Internal Storage. There is apparently a minimum size of internal storage necessary to scan and convert statements in a machine-independent language efficiently to the corresponding machine language program. In practice this has been found to be 2^{12} (=4096) words, each word handling a minimum of 6 characters. Storages from 2^{13} to 2^{15} in size are of course more advantageous. A storage of 2^8 is adequate for only the most ingenious scientific subroutines, wasting too much programmer effort to be useful for commercial work.

5.13 External Storage. The lack of medium access, medium cost storage media such as magnetic tape is an example of a machine characteristic which severely limits conceptualization of better computer usage. Although magnetic tape is for linear files, which have certain computational drawbacks, it is exceptionally useful for supervisory control and library facilities in an integrated system of data processing. This narrowness of conception is particularly evident in England, where tape usage is limited. A 1959 survey showed that only 11 out of 69 commercial computer systems were equipped with magnetic tape [47]. Few British programming systems actually control computer action automatically over multiple problems [48, 49]. Even when synthetic language is mechanically translated into machine language, corrections to the running programs are usually still made in machine language [50]. External storage media like tapes are mandatory for the use of application library techniques.

If such executive control is common in USA, it is not because the users are cleverer, but rather because the very existence of tape units in volume has prompted such experimental usage and development. In a survey of 61 large computers [32], government equipment averaged 18 tape units per machine, nongovernment equipment averaged 13 units. In both cases three units were used for peripheral operations. Each unit is capable of holding 5,000,000 characters per reel on the average, but there are some short tapes. Including metal, acetate and "Mylar" tapes, there are over 600,000 reels of tape in USA today.

As the design of modern architecture would not have been possible with the structural materials of a decade ago, it required the availability of magnetic tape in quantity to trigger and inspire new systems concepts and designs.

5.14 Instruction Repertoire. An examination of early programs for small internal storages show complex modification of instructions through looping and initialization. Present machines have larger storage and, perhaps more importantly, instructions which utilize index registers and indirect addressing. Not only do these features reduce the number of instructions necessary to do complex procedures, but they also reduce the amount of error which may be introduced in the program to be corrected later. It is safe to say that less than 10 per cent of all program instructions are ever modified today, over the entire spectrum of problems. In commercial applications alone, it is probably less than 5 per cent.

This characteristic may lead to permanent read-only memory [51] and larger programs with fewer loops. For example, the introduction of a photographic plate containing the entire basic programming system would have a heavy effect upon application programming. One of the present problems is to contain the working program so that incorrect modifications will not destroy the operating system with all its linkages to necessary auxiliary routines. Some present computers have provision for programmed storage protection by blocks to avoid such difficulties [52]. This would not occur with the programming system in separate storage from the working program.

5.2 Cooperative Organizations

One of the mixed blessings of computer usage is the ability of the machine designer to outstrip the last model by a factor of ten or so. The programmer and user is not susceptible to such magnification without artificial aids. For a single machine not much can be done, but for a group of identical machines the costs of programming can be spread out and amortized.

Early in 1954 a group of aircraft companies in USA found, in planning to replace IBM 701's by 704's, that severe dislocation of production would occur during the changeover by virtue of the reprogramming necessary even though the machines had common generic characteristics. It was found upon examination that a vast amount of duplication and redundancy had existed in the usage of the earlier machine. The question became *"Should basic programming remain in the realm of competitive advantage, or should a cooperative venture provide basic tools for all?"*

The outcome of this study was the SHARE organization, an informal cooperative among 704 users that has since grown to well over 100 members, each with at least one 704 installed or on order. It has been expanded since to include the successors 709 and 7090 as well. How well this organization succeeded is indicated by comparing the number of programming systems for less than twenty 701's with the number of systems for over one hundred 704's [53]. Within a general framework of assignment, each installation contributes basic programs with prescribed documentation to the entire body to use or modify as they wish. Accompanying each program, however, is a disclaimer that frees the originator of legal responsibility for its correct operation.

Following this single and successful venture, insularity disappeared in many areas. Functioning user groups include:

Group	*Machines*
ALWAC Users Association	ALWAC III, IV, V
CO-OP	CDC 1604
CUE	Burroughs 220
DATAMATIC 1000 Users Group	Datamatic 1000
DUO	Datatron 201 to 205
EXCHANGE	Bendix G-15
FAST	U.S. Army Fieldata Equipment
GUIDE International	IBM 705, 7070, 7080
LINC	Sperry Rand LARC
MCUG	Military AN FSQ series
PB 250 Users Group	Packard Bell 250
POOL	Royal-McBee LGP-30
RCA 501 Users Group	RCA 501
RUG	Autonetics RECOMP II
SHARE	IBM 704, 709, 7090
TUG	Philco 2000 (formerly Transac)
UNIVAC Users Group	UNIVAC Tape Systems
USE	UNIVAC Scientific 1103 and 1105

The above groups are all oriented to specific machines. In addition there are other groups oriented to particular applications or disciplines. They are:

Group	*Orientation*
CAMP	Military Applications
HEEP	Highway Engineering Exchange
NCG	Nuclear Codes Group
POUCHE	American Inst. of Chemical Engineers
ZMMD	ALGOL (Zürich-Mainz-Munich-Darmstadt)

These groups have found by experience that basic programming (the education of the machine) is not a competitive advantage after all, for each member has acquired a more intelligent machine for his particular applications through joint effort. This emphasizes that a certain amount of basic education is vital to operate a computer with any efficiency, whether it be for a single machine or a hundred like it. The original computer and the original programming may cost a million dollars each. The second computer costs nearly as much, but the second and succeeding sets of programs are available at only the cost of reproducing some cards or magnetic tape.

The success of organizations of this type in promoting operational standards has been marked. They also serve as a unified source of feedback for marketing criteria and information to the manufacturer. Interchange of new ideas and methods has effectively seeded and lifted the level of technical competence far above what might be accomplished by the secretive or insular user. Gone are the days when one oil company refused to test its programs on the manufacturer's sample machine for fear another oil company might steal its secrets and methods by a storage dump.

At present, the only user organizations existing outside USA are ZMMD and GUIDE International (including the Committee for Europe), which has over 230 participating installations.

These organizations have strong control over specifications of operating systems. The SHARE group, after selecting and improving SAP, the standard assembly program for the 704, completely specified an extensive operating system called SOS for the 709. (Cf. [54 to 59].) Gradually the interchange of programs is moving from those written in machine-oriented assembly language to those written in procedure-oriented and machine-independent languages such as ALGOL and COBOL. This ensures usage both to the next generation of computer for that group and, in many cases, exchange between several user groups.

A majority of these user groups have formed a joint users group, called JUG. A loose affiliation with the Association for Computing Machinery was accomplished in May 1961.

5.3 Standardization

5.31 Programming Languages. Much of the evolution of synthetic machine-independent languages has been quite similar. Most of the original translators for algebraic languages evolved roughly in the same era (cf. refs. [60 to 64]). The ALGOL 60 language is notable for the adaptation by P. Naur of the meta-linguistic symbology of J. W. Backus [65], an entire department in the journal *Communications of the ACM* devoted to algorithms written in ALGOL, and the series of textbooks in ALGOL planned by Springer Verlag, Berlin, Germany.

Standardization in scientific languages preceded that in commercial languages, just as scientific usage of computers preceded commercial usage in volume. No professional body such as ACM or GAMM took equal interest in the problem of commercial data processing languages, possibly because the problems were more difficult. In the absence of any requested action, the U.S. Department of Defense convened a meeting of manufacturers and users on May 28 and 29, 1959 to consider such an effort. Committees were established for short range, intermediate and long range considerations. In particular, the short range committee was asked to prepare a proposal for a blend of FLOW-MATIC, AIMACO [66] and Commercial Translator by September 1959. This was to serve as a stopgap language which could be useful for a period of two years until supplanted by the language to be developed by the intermediate group.

British manufacturers took an extreme interest in this effort and were called together by International Computers and Tabulators (whose corresponding language was CODEL) [67] in July 1959 to consider the same problem. It was decided to await results from the group in USA and then evaluate that language.

As it developed, the short range group of CODASYL (Committee On Data Systems Languages) preempted the domain of the intermediate group, which was canceled. The resulting language was called COBOL (COmmon Business Oriented Language) [13] and went somewhat further than the original directive called for. The language is complex and conditions are worsened by some unreconcilable differences in various equipments. Despite some remaining flaws and differences in reconciliation, the following manufacturers have announced COBOL processors for the indicated machines:

Manufacturer	*Machine*
Bendix	G-20
Burroughs	B5000
Control Data	1604
Minneapolis-Honeywell	400
Minneapolis-Honeywell	800
Philco	2000
General Electric	225
IBM	705 III/7080
IBM	7070/7074
IBM	709/7090
IBM	1401
IBM	1410
ICT	1301
NCR	304
NCR	315
RCA	301
RCA	501
RCA	601
Sperry Rand	UNIVAC II
Sperry Rand	UNIVAC III
Sperry Rand	490
Sperry Rand	1107
Sperry Rand	SS80/SS90
Sylvania	MOBIDIC

5.32 Systems Standards. The chief obstacle to writing a single program for all different machines has been the intractability of hardware design. Many aspects of computer design must reflect competitive technologies and salable characteristics. However, many differences between the several computers have been, in the words of J. C. McPherson, *"capricious and arbitrary"*. Many different options may be equally suitable, but when differing options are selected through noncognizance – it is time for standards organizations to step in. It is possible that this area will contribute heavily to the ιeduction and simplification of programming effort.

A joint project in the standardization of certain aspects of data processing equipment has been formed with TC97 (Technical Committee 97) of the ISO (International Standards Organization) and TC53 of the IEC (International Electrotechnical Commission). Initial work will proceed in four areas, commencing with the first meeting in Geneva in May 1961. These areas are:

Character Sets and Representations
Data Transmission
Programming Languages
Glossary of Terminology

The first two areas have to do with the common language interchange of both data and programs between users and various equipments. It has been found that much of the complexity in COBOL and similar languages is due to the need to take care of basic differences in this area. The third area implies that there may be an eventual joining of the scientific and commercial procedure languages. This is supported by two trends noticed by workers in the field:

(a) Properties formerly exclusive to either type of language are very useful to each other. The business language is enhanced by algebraic notation and subscripting, the algebraic language can be improved by separating out data description and being able to refer to operands other than floating point variables.

(b) The underlying syntactical structure of both types of languages is similar enough to suggest an eventual blending into a common language for all purposes, each with its own jargon or dialect, if necessary, but enough equivalent that common processors may be used for either.

Another factor in raising the efficiency of programming is the selection of standard machine configurations. User groups do this to limit the variety of programs needed. Although some of the variables in modular systems are compensated for by program generation (such as varying sizes of internal storage), it is generally advantageous to pick a specific configuration which is not always the minimum. For example, the first SHARE standard 709 specified a 8,192 word storage. However, it turned out that almost all machines were ordered with a 32,768 word storage because the cost of the additional storage was more than offset by the increased power in problems per dollar. Most programming systems are attuned to top efficiency for a particular configuration. Sometimes they are not even prepared for lesser configurations. It is usually advantageous in cost to get additional hardware to bring the configuration up to the standard because of the more than compensating savings achieved through use of the programming system.

5.4 Experience

S. Gill [68] states that *"the practical business of tapping the vast potentialities of computers has come as such a novelty to us that we are practically developing an entirely new subject — a new version of mathematics, if you like"*. Considering the astonishing rate of growth in programming, it is not surprising that the literature has not had a chance to catch up properly. Besides, programming more than nearly any other field is learned by doing rather than reading.

Without risking philosophical debate, programming may be said to have enough of the nature of thought processes that new developments stem mostly from circumstances and not from speculation. The most effective means of disseminating such acquired knowledge is by *seeding* less experienced groups with a few highly experienced people. This has been adequately demonstrated by programmers who, having reached a stasis point in one group, move to another group with a higher experience level and quickly develop to a corresponding position in that group.

Conversely, it has been noted that those programmers that advance to higher management positions (that do not involve actual contact with machines and methods) quickly fall behind current technology levels unless they make strong efforts to keep up with new techniques. The present era of programming is one of stumbling in search of complete concepts. Current theories of programming are faulty and conflicting. The programmer who stops now is likely to retain a useless orientation for the future.

5.5 Education and Literature

The extremely rapid growth of the computing field has caused a notable lag in the publications of timely papers in the technical journals and in program documentation. Actual practice has preceded in time the publication of the theory of practice to a surprising extent. Perhaps this has been due to a certain attitude of waiting to see if the field would achieve true professional status.

5.51 Universities. Although single universities (such as Manchester and Cambridge in England; Mainz in Germany; Michigan, Illinois, Princeton, UCLA, MIT and others in the USA) made developmental efforts in both hardware design and programming, the infant science of computing was attached to a variety of departments. Such work has been supported variously by departments of mathematics, business administration, electrical engineering and any other with enough funding and interest to nurture a beginning. To date, no university recognizes a chair in information processing, which is the general field encompassing the computer sciences.

Not until 1957 was there any general effort to train people for computer design and programming. Even here the universities did not take the lead by themselves. The manufacturers, extrapolating to a drastic situation in the expanding field, took steps to provide universities with special and production computers for training purposes. The effect is now being felt. A few universities stand out remarkably in programming. In Germany there are Munich, Mainz, and Darmstadt, in Switzerland there is Zürich, and the USA has Carnegie Tech, Case Institute, MIT and Michigan. England has relaxed the early lead in programming techniques taken by M. V. WILKES at Cambridge and R. A. BROOKER at Manchester.

The impact of programming training at universities is now felt. Each graduate from the Massachusetts Institute of Technology in 1961 will have taken a mandatory course in computer programming. The latest count shows a total of 118 computers in universities in North America [69]. There are approximately 65 computers in European universities. Pages 135—138 of [9] list 145 universities in USA offering courses in automatic data processing and systems.

5.52 Manufacturers. The education of the user is of extreme interest to the marketer of a product. Many manufacturers operate their own training schools in order to staff satisfactorily a large number of machines. Some of these schools are larger than universities. For example, IBM currently trains about 11,000 programmers a year as part of a general educational program which reaches over 120,000. A program on this scale is necessary to achieve a predicted work force of 170,000 professionals in 1966 for USA alone [70]. Although sheer volume does not necessarily produce improved methods, the net effect has been an accelerated learning process in the efficient utilization of machines through programming. The description and documentation of programming systems has become more professional.

Manufacturers also support the informal educational process by distribution of technical literature. One manufacturer distributed over 450,000 copies of more than 150 different publications in 1959 [71]. In this instance, distribution of exchange programs among members of user groups averaged 150 programs a day.

5.53 Teaching by Machine. One of the most promising methods for raising the level and competence of programming is to enlist the aid of the machine itself. The Computation Center at the Carnegie Institute of Technology, under A. J. Perlis, utilizes a 650 RAMAC to teach students to program that same machine. The student keypunches his name on a card, drops it in the read hopper and is automatically enrolled in the course. Provision is made for orderly progression through the lessons. When the lesson program written by the student does not work, the teaching program analyzes the faults and sends him back to restudy the proper previous lesson.

Many experiments are being made in automatic teaching by computer [72, 73]. North American Aviation has used semiautomatic methods to teach the FORTRAN language to over a thousand of its engineers. Computers are also being used to evaluate the effectiveness of programmers and point out where additional training or discipline is needed [74].

6. Costs and Statistics

6.1 Programming Systems

Some idea of the relative size of programming systems may be gained from the following survey (cf. [75]):

System	Machine	Number of Machine Language Instructions	System Type
SURGE	704	12,000	Sort, report generator
CLIP	709	18,000	Information processor
APT	704, 709	35,000	Machine tool language
CL–1	709	45,000	Information processor
SOS	709	50,000	Compiler, operating

A typical major programming system will cost from $ 250,000 to $ 1,000,000 to produce and require from 10 to 50 very high level programmers working from one to two years.

The average cost per instruction produced is much higher for programming systems than it is for applications, as they are much more complicated and generalized. The cost may vary from 13 to 25 dollars an instruction in order to produce a system which users may program at costs of only one or two dollars an instruction.

These investments are returned by the decrease in programming time. For example, a single Commercial Translator statement used in a payroll problem yielded 37 distinct machine language instructions, all correct without further diagnostics.

The costs of such programming systems may be expected to be reduced sharply in the next few years. In IBM's experimental expanded ALGOL system, for example, only 800 instructions are actually written in machine language; all others are written in expanded ALGOL itself and are thus usable for many different machines. The only machine language instructions needed are those for basic symbol manipulation and reduction to symbolic macro-instructions. The system is presently at about 12,000 machine language instructions; therefore 11,200 of these have been self-generated.

6.2 Programs for Specific Applications

6.21 Size. One of the largest applications on record [76] requires 65 separate machine runs for a single problem. With an average of 3,000 machine instructions per storage fill, this gives a total program size of about 200,000 instructions.

An oil company's first nine programs written in 705 Processor language [77] averaged 2760 instructions per program, or 13,800 characters. The total programs required from 7500 to 36,000 characters of storage, averaging 20,000.

6.22 Instruction Cost. Surveys taken in 1957 yielded the following average costs per checked out instruction:

Language	Cost per Instruction ($)
Machine language	10
Symbolic assembly	5–6
Symbolic + macros	2–3
Independent language	1

Further statistics are available for the programs mentioned in Section 6.21. The average times for the nine programs were:

Block diagram, code, assembly	7 days programmer time
Assembly	97 minutes (avg. 2.5 assemblies per program)
Machine test	50 minutes

A rough calculation with these data yields less than a dollar per checked out instruction, quite comparable to that for machine-independent languages.

The cost of moving applications to different machines varies considerably with the source language used to write the programs. Table 1 shows the additional advantages accruing from each additional degree of machine independency. Thus, machine-independent languages are extremely useful not only as an aid in decreasing the original cost of programs but also as insurance against moving the program to different machines.

6.23 Staff. The largest computers, depending upon the class and variety of applications, may require a staff of from 50 to 75 people [78].

Table 1. Comparison of typical conversion from one machine (family) to another

Program Written in	Effort for Machine "A"		% Additional Effort for Machine "B"		Net Additional Effort for Machine "B"
Machine Language	100 % (base)	×	100 %	=	100 %
Symbolic Assembly Language	80 %	×	80 %	=	64 %
Macro-Language	60 %	×	40 %	=	24 %
Procedure Language	20 %	×	25 %	=	5 %

Bibliography

[1] Gotlieb, C C.: General-Purpose Programming for Business Applications. In: Advances in Computers, Vol. 1 (Ed.: F. L. Alt). Academic Press, New York 1960, pp. 1—42.

[2] McCracken, D. D., Weiss, H., Lee, T.-H.: Programming Business Computers. J. Wiley & Sons, New York 1959.

[3] McCracken, D. D.: Digital Computer Programming. J. Wiley & Sons, New York 1957.

[4] Gotlieb, C. C., Hume, J. N. P.: High Speed Data Processing. McGraw-Hill, New York 1958.

[5] Jeenel, J.: Programming for Digital Computers. McGraw-Hill, New York 1959.

[6] Wrubel, M. H.: A Primer of Programming for Digital Computers. McGraw-Hill, New York 1959.

[7] Leveson, J. H. (Editor): Electronic Business Machines. Heywood & Co., London 1959.

[8] Ledley, R. S., Justed, L. B.: Computers in Medical Data Processing. Operations Research **8** (1960) No. 3, pp. 299—310.

[9] 86th Congress, Use of Electronic Data Processing Equipment. U.S. Government Printing Office (41783), Washington, D.C. 1959.

[10] The Preparation of Macro-instructions for Use with Autocoder III. Reference Manual C 28—6056—1, IBM Corp., New York 1959.

[11] Programming the IBM 705 Using the Autocoder III System. Reference Manual C 28—6057, IBM Corp., New York 1959.

[12] Williams, F. A.: Handling Identifiers as Internal Symbols in Language Processors. Communications ACM **2** (1959) No. 6, pp. 21—24.

[13] Conference on Data Systems Languages (CODASYL), COBOL, Initial Specifications for a COmmon Business Oriented Language. U.S. Government Printing Office, Washington, D.C. 1960.

[14] UNIVAC FLOW-MATIC Programming System. Reference Manual U—1518, Sperry Rand Corp., New York 1958.

[15] Greenwald, I. D.: A Technique for Handling Macro Instructions. Communications ACM **2** (1959) No. 11, pp. 21—22.

[16] Case Study Solutions for the 705 Autocoder. Form 22—6740—0, IBM Corp., New York.

[17] IBM Commercial Translator. General Information Manual F28—8043, IBM Corp., New York 1960.

[18] FACT Manual. Report 160—2M, DSI—27, DATAmatic Division. Minneapolis-Honeywell, Wellesley Hills, Mass. 1960.

[19] Goldfinger, R.: Syntactical Description of a Common-Language Programming System. CODASYL Report, March 8—10, 1960.

[20] APT, The Automatically Programmed Tool System (7 Volumes). Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass. 1959.

[21] Lo, Y. C., Gaudette, C. H.: An Automatic Multiprogram Operating System. Preprint 60—329, Amer. Inst. Electr. Engrs., New York, February 1960.

[22] Smith, R. B.: The BKS System for the Philco 2000. Communications ACM **4** (1961) No. 2, pp. 104, 109.

[23] CAOS, Completely Automatic Operational System. Report LMSD—48482, Lockheed Aircraft Corp., Sunnyvale, Cal. 1959.

[24] Introduction to the CL—1 Programming System. Manual TR 59—6, Technical Operations, Inc., Washington, D. C. 1960.

[25] Bemer, R. W.: A Checklist of Intelligence for Programming Systems. Communications ACM **2** (1959) No. 3, pp. 8—13.

[26] Kavanagh, T. F.: TABSOL — A Fundamental Concept for Systems-Oriented Languages. Proc. Eastern Joint Computer Conf., New York, Dec. 13—15, 1960, pp. 117—136.

[27] Evans, O. Y.: Advanced Analysis Method for Integrated Electronic Data Processing. Bulletin F20—8047, IBM Corp., New York 1960.

[28] Young, J. W., Kent, H. K.: Abstract Formulation of Data Processing Problems. J. Ind. Engng. **9** (1958) No. 6, pp. 471—479.

[29] Bemer, R. W.: Survey of Modern Programming Techniques. Computer Bulletin **4** (March 1961) No. 4, pp. 127—135.

[30] Wolpe, H.: Algorithm for Analyzing Logical Statements to Produce a Truth Function Table. Communications ACM **1** (1958) No. 3, pp. 4—13.

[31] The Auditor Encounters Electronic Data Processing. Price Waterhouse & Co., New York 1958.

[32] Proceedings of GUIDE, X, May 1960.

[33] Betz, B. K., Carter, W. C.: New Merge Sorting Techniques. Preprints, 14th ACM Conference, Massachusetts Institute of Technology, Cambridge, Mass., Sept. 1—3, 1959.

[34] Batchelder, J. C.: Sorting Methods for IBM Data Processing Systems. General Information Manual F28—8001, IBM Corp., New York 1958.

[35] McGee, R. C., Tellier, H.: A Re-Evaluation of Generalization. Datamation **6** (1960) No. 4, pp. 25—29. Cf. also: User's Reference Manual, 9 PAC System, IBM Corp., New York 1960.

[36] IBM 7070 Report Program Generator. Bulletin J28—6049, IBM Corp., New York 1959.

[37] Report Program Generator for IBM 1401 Card Systems. Bulletin J29—0215, IBM Corp., New York 1960.

[38] The Chaining Method of Disk Storage Addressing for the IBM RAMAC 305. Bulletin J28—2008—1, IBM Corp., New York 1958.

[39] The Chaining Method for the 650 RAMAC System. Bulletin J28—4002, IBM Corp., New York 1958.

[40] Peterson, W. W.: Addressing for Random Access Storage. IBM Journal Res. & Dev. **1** (1957) No. 2, pp. 130—146.

[41] Johnson, L. R.: An Indirect Chaining Method for Addressing on Secondary Keys. Communications ACM **4** (1961) No. 5, pp. 218—223.

[42] Bemer, R. W.: Do It By the Numbers (Digital Shorthand). Communications ACM **3** (1960) No. 10, pp. 530—536.

[43] Ershov, A. P.: Programming Programme for the BESM Computer. Pergamon Press, Oxford 1959. (Transl. from Russian.)

[44] Ljapunov, A. A.: On Logical Schemes of Programs (in Russ.). Problemi Kibernetiki Vol. 1. State Publishers of Physico-Mathematical Literature, Moscow 1958, pp. 46—74. (Engl. transl. in preparation by Pergamon Press, Oxford.)

[45] Smith, H. J.: A Short Study of Notation Efficiency. Communications ACM **3** (1960) No. 8, pp. 468—473.

[46] Bemer, R. W.: Survey of Coded Character Representation. Communications ACM **3** (1960) No. 12, pp. 639—641.

[47] Williams, R. H.: The Commercial Use of Computers in Britain. Automatic Data Processing **1** (Nov. 1959), pp. 33—35.

[48] Cook, R. L.: Time-Sharing on the National-Elliott 802. Computer Journal **2** (1960) No. 4, pp. 185—188.

[49] Strachey, C.: Time Sharing in Large Fast Computers. Computers and Automation **8** (Aug. 1959) No. 8, pp. 12—16.

[50] Gill, S.: Current Theory and Practice of Automatic Programming. Computer Journal **2** (1959) No. 3, pp. 110—114.

[51] Atlas (Ferranti Ltd., Manchester and London, England). Digital Computer Newsletter **12** (1960) No. 4; reprinted in: Communications ACM **3** (Oct. 1960) No. 10, pp. 580—582.

[52] Nekora, M. R.: Comment on a Paper on Parallel Processing. Communications ACM **4** (1961) No. 2, p. 103.

[53] Bemer, R. W.: The Status of Automatic Programming for Scientific Problems. Proc. 4th Annual Computer Applications Symposium, Armour Research Foundation of Illinois Institute of Technology Chicago, Ill., Oct. 24—25, 1957, pp. 107—117.

[54] Boehm, E. M., Steel, T. B.: The SHARE 709 System, Machine Implementation — Symbolic Programming. Journal ACM **6** (1959) No. 2, pp. 134—140.

[55] Bratman, H., Boldt, I. V.: The SHARE 709 System, Supervisory Control. Journal ACM **6** (1959) No. 2, pp. 152—155.

[56] Digri, F. J., King, J. E.: The SHARE 709 System, Input-Output Translation. Journal ACM **6** (1959) No. 2, pp. 141—144.

[57] Greenwald, I. D., Kane, M.: The SHARE 709 System, Programming and Modification. Journal ACM **6** (1959) No. 2, pp. 128—133.

[58] Mock, O., Swift, C. J.: The SHARE 709 System, Programmed Input-Output Buffering. Journal ACM **6** (1959) No. 2, pp. 145—151.

[59] Shell, D.: The SHARE 709 System, A Cooperative Effort. Journal ACM **6** (1959) No. 2, pp. 123—127.

[60] Rutishauser, H.: Über automatische Rechenplananfertigung bei programmgesteuerten Rechenmaschinen. Z. angew. Math. Mech. **31** (1951), p. 255.

[61] Rutishauser, H.: Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen. Mitt. Inst. Angew. Math. ETH Zürich, No. 3. Verlag Birkhäuser, Basel 1952.

[62] Laning, J. H., Zierler, N.: A Program for Translation of Mathematical Equations for Whirlwind I. Engineering Memorandum E—364, Massachusetts Institute of Technology, Cambridge, Mass., January 1954.

[63] Programmers Reference Manual FORTRAN. Internat. Business Machines Corp., New York 1956.

[64] Brooker, R. A.: Some Technical Features of the Manchester Mercury Autocode Programme. Symposium Mechanisation of Thought Processes, National Physical Lab., Teddington, Nov. 24—27, 1958, pp. 201—229.

[65] Naur, P. (Editor), et al.: Report on the Algorithmic Language ALGOL 60. Numerische Mathematik **2** (1960) No. 2, pp. 106—136; reprinted in: Communications ACM **3** (1960) No. 5, pp. 299—314.

[66] AIMACO, The Air Material Command Compiler. Manual AMCM 171—2, Wright-Patterson Air Force Base, Ohio 1959.

[67] Wensley, J. H., et al.: An Introduction to the CODEL Automatic Coding System. Computer Developments Ltd., Kenton, Middlesex 1959.

[68] Gill, S.: The Philosophy of Programming. Annual Review in Automatic Programming, Vol. 1 (Ed.: R. Goodman). Pergamon Press, Oxford 1960, pp. 178—188.

[69] Reeves, R. F.: Digital Computers in Universities I—IV. Communications ACM **3** (1960) No. 7, p. 406; No. 8, p. 476; No. 9, p. 513; No. 10, pp. 544—545.

[70] Special Report on Computers. Business Week, June 21, 1958.

[71] Jones, G. E.: Address to the International Systems Meeting, October 12, 1960, New York City.

[72] Bauer, W. F., Gerlough, D. L., Granholm, J. W.: Advanced Computer Applications. Proc. IRE **49** (1961) No. 1 (Computer Issue), pp. 296—304.

[73] Galanter, E. (Editor): Automatic Teaching. J. Wiley & Sons, New York 1959.

[74] Boguslaw, R., Pelton, W.: STEPS — A Management Game for Programming Supervisors. Datamation **5** (1959) No. 6, pp. 13—16.

[75] Wagner, F. V.: Summary of Questionnaire on New POLs. SHARE, February 15, 1960.

[76] Proceeding of GUIDE, II, 1957.

[77] The IBM 705 Processor. Bulletin J28—6068, IBM Corp., New York 1959.

[78] Paine, R. M.: Selection of Computer Personnel. Computer Bulletin **3** (1959) No. 2, pp. 23—26.