Compiling Techniques for Algebraic Expressions

By Harry D. Huskey

This paper describes a method of translating algebraic formulae into a computer program in a single pass.

1. Introduction

Various algebraic and algorithmic languages have been proposed as a means to communicate with high-speed digital computers. There has been some discussion of techniques of processing algebraic statements. The following paper presents a method of translating from algebraic formulas into computer commands in a single pass, processing the formula from left to right. The system takes into account parenthesis, subscripted variables, and real functions of a single real variable. Extensions to functions of several variables (or procedures) are clearly possible.

In order to present the technique it is necessary first to present an object computer language which will be the output language of the system. The items named in the formulas will be denoted by single letters of the alphabet. Extensions to multi-symbol names and to numbers are obviously possible.

Next, the specifications of the input language will be given. In order to describe the translation process from the input language to the output, or object language, a list of terms and symbols will be defined, and the language used to describe the translation process will be presented. Finally, the actual translation algorithm will be given.

2. Object Language

The commands in the object language will have the structure of operation code and address.

The address is an integer, a, in the range $0 \leqslant a \leqslant 9999$ say, depending on the size of the memory. The object computer considered here is of the simplest kind. More efficient object programs may be obtained by similar techniques on computers with more facilities.

Table 1 gives the operation codes of the object language. This is a partial list of commands for a digital stored-program computer. Only those commands needed are given; for example, input and output commands, or transfer of control commands, are not given. The arithmetic commands are designed to work in a floating-point number system. Note that there are two kinds of division.

The accumulator index command causes the contents of the accumulator to be rounded to the nearest integer and the result added to the address of the next following command. Thus, the accumulator can play the role of an index register.

The DO and RETURN (RE) commands provide for the use of subroutines or functions. A DO command implies that when the specified task is completed the computation is to continue at the current location. For example, if a function symbol appears in the middle of a formula (which implies that the function has been defined somewhere else) the DO command provides a means of transferring control to the subroutine which defines the function. After the function is computed the computation will continue processing the current formula. In order to accommodate functions of functions with multiple arguments, etc., an entry is made in a return list for each DO operation. When compiling

Table 1

Object Computer Command List

Floating-point Commands		
CA	Clear and Add	$(a) \rightarrow (A)$
AD	Add	$(A) + (a) \rightarrow (A)$
CS	Clear and Subtract	$-(a) \rightarrow (A)$
SU	Subtract	$(A) - (a) \rightarrow (A)$
MU	Multiply	$(A) \times (a) \rightarrow (A)$
DI	Divide	$(A)/(a) \rightarrow (A)$
ID	Inverse Divide	$(a)/(A) \rightarrow (A)$
ST	Store	$(A) \rightarrow (a)$
NE	Negate	$-(A) \rightarrow (A)$
Index Command		
AI	Accumulator Index	(A) added to address of next command.
DO and RETURN Commands		
DO	Do (function, subroutine or procedure)	If the command is in location L then $L + 1$ is stored in first free space of the return list, and control is transferred to a.
RE	Return	a is added to the last entry in the return list and control is transferred to the resulting location. This last entry is deleted from the return list.

Note: See Table 3 for symbols and abbreviations.

(translating) a formula which defines a function, a RETURN command is added to the object program at the conclusion of the defining program.

The effect of the DO and RETURN commands is given in detail in Table 2. For example, for DO commands the number in the command counter (CC) is increased by unity and stored in the memory location specified by the contents of the return counter (RC). The number in the return counter is then increased by unity, and the address a of the current command (in the command register, CR) is transferred to the command counter.

3. The Input Language

The items involved in the process will be named using single letters from the alphabet, and will be referred to as *variables*. If X and Y are any variables, then $X(Y)$ denotes a variable and, depending upon context, may mean the subscripted quantity X_Y (implying that Y is integral valued or, if not, that the nearest integer to Y is to be used), or it may mean the function X of the variable Y. However, to simplify this presentation, subscripted quantities will be denoted as $X[Y]$.

Variables and arithmetic symbols may be used to make up expressions according to the following rules:

3.1 *Expressions*

1. If E and F are expressions and R is an arithmetic operator $+$, $-$, ., or /, then ERF is an expression, provided that the first symbol of F is not "$+$" or "$-$".
2. If E is an expression then (E) is also an expression.
3. If E is an expression and its first symbol is not an arithmetic operator, then $+E$ and $-E$ are expressions.
4. Any variable, subscripted variable, or function is an expression. (Here function is limited to a real function of a single real variable.)

3.2 *Statements*

A computation is defined by a sequence of *statements*. If V is a variable and E is an expression (not a variable) and p a punctuation symbol, then $V = Ep$ and $E = Vp$ are statements defining the value of V. These are read as: "V is replaced by (the value of) E" and "(the value of) E replaces V", respectively. If E is also a variable, then the right replaces the left. Functions are defined as follows: $F(X) = Ep$ or $E = F(X)p$, where E is an expression (usually involving X) which defines the value of F for each X. Definitions of functions are also called *statements*.

4. The Descriptive Language

The input language may be extended so as to provide an efficient means of specifying the translation process. Thus, the descriptive language includes the concepts of variables, expressions, and statements as described in the section on input language.

Table 2

Subroutine Transfers

DO: (CC) + 1 → ((RC)), (RC) + 1 → (RC), a → (CC).
RE: ((RC)) + a → (CC), (RC) − 1 → (RC).

Note: (CC) denotes the item whose name is "CC", or the contents of register CC.

4.1 *Names*

Variables are *names*. Furthermore, letters of the alphabet and digits may be concatenated to form other names.

If S is a statement, N a name, and p a punctuation symbol (comma, semi-colon, or period), S may be given the name N by writing "$N{:}Sp$".

4.2 *Indices*

The range of an index I used with subscripted variables may be specified by writing $I = E(F)Gp$, where E, F, and G are expressions. $I = E(F)Gp$ means that I is to take on the values of E, $E + F$, $E + 2F$, . . . , $E + kF$,

where $E + kF \leqslant G < E + (k + 1)F$,

or $E + (k + 1)F < G \leqslant E + kF$.

4.3 *Braces*

If S_i are statements for $i = 1(1)n$, then $\{S_1, S_2, \ldots, S_n\}$ is a statement. If statements S involve subscripted variables with subscript I they may be executed for all the specified values of I by writing $\{S\}I = E(F)Gp$ (see Section 4.2).

4.4 *Integers*

A concatenation of digits may be used like variables with the usual meaning.

4.5 *Arithmetic Statements*

In the descriptive language arithmetic statements will take the form "$E \rightarrow Vp$" instead of "$E = Vp$" as in the input language. This illustrates the independence of the two and the versatility of the algorithm.

4.6 *Imperative Statements*

The statement "ABC", is an instruction to "DO" or "EXECUTE" the statement named ABC. The range or scope of such a "DO" instruction is from the specified name to the first period, with the body of the "subroutine" being enclosed in braces.

The statement "ABC" has the effect of a transfer of control. The execution moves to the statement labelled ABC and continues there with no *a priori* expectation of return to whatever follows the period of "ABC".

4.7 *Decision Statements*

If E is an expression, R a relational symbol $<$, $\leqslant$, $=$, $\geqslant$, $>$ or $\neq$, V is a variable, T, U, X, W are statements, and p is a punctuation symbol (comma, semi-

colon, or period), then *ERV*? *Tp Up Xp Wp* . . . is a *decision statement*. If the relation *R* is true, statements *T*, *X*, . . ., etc., will be executed. If the relation is false, statements *U*, *X*, *W*, etc., will be executed. That is, if true, the statement between the question mark and the first punctuation is executed, otherwise the statement between the first *p* and the next *p* will be executed. In either case continuation is with statement *X*.

5. Auxiliary Features of the Translator

In order to translate formulas into an object program it is necessary to get the symbols which comprise the formulas into the computer, and the generated commands must either be stored somewhere in the memory or recorded (on paper tape, cards, or magnetic tape) for subsequent loading into the computer at running time. The precise structure of these auxiliary programs is not pertinent to this presentation; such routines will therefore be presented with a minimum of detail.

5.1 *Input and Select Next Symbol* (SNS) *Routine*

It will be assumed that a program exists which will load a statement into the computer memory.

The *select next symbol* (SNS) routine replaces NS (next symbol) by the next following symbol in the formula. If the formula is referred to as the symbol list (SL) and a symbol counter (SC) specifies which symbol is currently under consideration, then the *select next symbol* routine could be written as "SNS : {SL(SC) → NS, SC + 1 → SC}.". That is, the symbol in the symbol list at SC replaces NS (next symbol), and the symbol counter (SC) is increased by unity. In most computers this routine would be more complicated than indicated above, in order to allow the packing of several symbols into a computer word.

5.2 *Output*

Whenever a command is generated it will be recorded in the object program (OP) at "location" CC (command counter). This "recording" may be a simple storage of the command in a portion of the computer memory at a location corresponding to CC. In practice this may be a punching of the command on to tape or cards. If T denotes the command, this recording process may be written "NC(T) : {T → OP(CC), CC + 1 → CC}.", where NC stands for "next command in the object program".

5.3 *Distribution on Symbol Pairs*

The translation routine considers pairs of symbols in the compiling process. One of these symbols is called the *next symbol* (NS) which is found by the SNS routine discussed above. The other symbol is either the *present symbol* (PS) or the *present operator* (PO).

If the portion of the formula currently under consideration is

$$\ldots + a/ \ldots$$

and that which precedes the "+" has just been processed, then NS is "+". In the course of processing the above portion, NS will be successively "*a*" and "/". PS will successively be "+" and "*a*", and PO will be "+". The changes are caused by the *advance routine* (ADV) and the *operand advance routine* (OADV). After processing a pair of successive operators (such as + and / in the above example), the OADV routine comes into action.

$$\text{OADV: } 0 \rightarrow A,\ \text{NS} \rightarrow \text{PO, ADV.}$$
$$\text{ADV: NS} \rightarrow \text{PS, SNS, PSNS.}$$

OADV sets the address "*A*" to zero, and transfers NS to be PO, then enters the advance routine (ADV). The advance routine transfers NS to PS, finds the next symbol (NS) by "doing" the select next symbol routine (SNS). Then there is a transfer to one of 81 "generators" (see Table 7) depending upon the symbol pair. This distribution is accomplished by the PSNS routine. One way of doing this is to look up in a code list (CL) a digit from 0 to 8, called the *distribution constant*, which corresponds to the row in which the symbol occurs in Table 7. Assume the column headed "+ −" is separated into two columns; then this routine could appear as "PSNS: 9 × CL(NS) + CL(PS) + 1 → T, T.", where, for example, CL(+) = 4, CL(*a*) = 0, and CL(/) = 7.

6. Memory Organization in the Object Computer

The commands of the object program will be placed in the first part of the memory, starting at location zero, and space for data and variables is assigned at the end of the memory, working backwards.

Thus, the command counter (CC) starts at zero and increases by unity for each command in the object program. A memory address (MA) starts at the maximum address for the memory (9999) and always denotes the "next available memory location". As each variable occurs for the first time, the next available memory location (MA) will be entered in the name list (NL) at the position which corresponds to the symbol, and MA will be reduced by unity.

7. Flags, Lists, and Subroutines

The meaning of symbols used in the Compiler specification are given in Table 3. In the translation the present system generates two lists while compiling: *name list* (NL) and the *temporary command list* (TL). The commands of the object program are recorded (punched) as they are generated. The object program deals with three "lists": the data memory, working storage, and the object program itself. The indices MA (memory address), WA (working address), and CC (command counter) refer to items in these respective lists.

The development and use of the name list is not an essential part of the discussion given in this paper. It is sufficient to assume that the name of each item mentioned is entered in this list, as well as an associated memory address and other information characterizing

Table 3

Symbols and Abbreviations

ABBREVIATION	DESCRIPTION	DEFINED IN	ABBREVIATION	DESCRIPTION	DEFINED IN
a	Object Computer Address	Sec. 2	NAM	Name Routine	Sec. 7
A	Address	Sec. 5.3	NC	Next Command Routine	Sec. 5.2
AC	Accumulator Flag	Sec. 9	"NE"	Negate Command	Table 1
"AD"	Add Command	Table 1	NL	Name List	Sec. 7
ADD	Add Routine	Table 6	NP	Name Position	Sec. 7
ADR	Address Routine	Table 6	NS	Next Symbol	Sec. 5.1
ADV	Advance Routine	Table 6	OADV	Operator Advance Routine	Sec. 5.3
"AI"	Accumulator-Index Command	Table 1	OP	Object Program	Sec. 5.2
			p	Punctuation symbol	Sec. 4.1
Braces	Statement Parenthesis	Sec. 4.3	period	Imperative Statement	Sec. 4.6
"CA"	Clear Add Command	Table 1	PO	Present Operator	Sec. 5.3
CAD	Clear Add Routine	Table 6	PS	Present Symbol	Sec. 5.3
CAST	Clear Add—Store Routine	Table 6	PSNS	Present Symbol—Next Symbol Routine	Sec. 5.3
CC	Command Counter	Sec. 2 or 5.2			
CD	Conditional Add Routine	Table 6	RC	Return Counter	Sec. 2
CG	Command Generator	Table 6	"RE"	Return Command	Table 1
CL	Code List	Sec. 5.3	S	Sign Flag	Sec. 9
Comma	Imperative Statement	Sec. 4.6	SC	Symbol Count	Sec. 5.1
CR	Command Register	Sec. 2	SF	Subscript Flag	Sec. 11
"CS"	Clear Subtract Command	Table 1	SL	Symbol List	Sec. 5.1
"DI"	Divide Command	Table 1	SNS	Select Next Symbol	Sec. 5.1
"DO"	Subroutine Execution Command	Sec. 2	STORE	Store Routine	Table 6
			T	Compiler Temporary Storage	Table 6
ERR	Error Routine	Sec. 9			
FUN	Function Routine	Table 6	TC	Temporary Command Routine	Table 6
GO TO	Imperative Statement	Sec. 4.6			
"ID"	Inverse Divide Command	Table 1	TCNC	Temporary Command—Next Command Routine	Table 6
Indices	Indices	Sec. 4.2			
INIT	Initialize Routine	Table 6	TL	Temporary List	Sec. 7
KN	Name List Limit	Sec. 7	TST	Temporary Store Routine	Table 6
L	Level	Sec. 8	U	Compiler Temporary Storage	Table 6
MA	Memory Address	Sec. 6	WA	Object Program Working Address	Table 6
"MU"	Multiply Command	Table 1			
MUID	Multiply—Inverse Divide Routine	Table 6	ZADV	Zero Advance Routine	Table 6

the item. Each entry in NL uses four memory locations. The procedure for looking up an item A in this list, NL, might be

$$\begin{aligned}&\text{NAM: } \{\text{KN} \to \text{K, NAM1: NL(K)} = A\,?\\&\qquad \{\text{NL(K} + 1) \to A,\ \text{K} \to \text{NP.}\},\\&\qquad \{\text{K} - 4 \to \text{K} \geqslant 0\,?\quad \text{NAM1.,;}\};\\&\text{KN} + 4 \to \text{KN} \to \text{NP},\ A \to \text{NL(KN)},\\&\quad \text{MA} \to \text{NL(KN} + 1) \to A,\ \text{MA} - 1 \to \text{MA}\}.\end{aligned}$$

where KN corresponds to the last entry in the name list, NP is the "name place," that is, the location in the name list of the last name considered. The portion of NAM preceding the ";" checks to see if the name is already in the list; the rest of the routine enters the name in the list. The first and third periods represent two exits to the routine. The braces enclosing the statements of the NAM routine indicate that it is a subroutine and is to be entered with a DO instruction.

Generally, the translation proceeds paying attention to only three symbols, NS, PS, and PO (sometimes PO and PS refer to the same symbol). Sometimes more attention must be paid to context. The compiler prepares

for such situations by noting the occurrence of certain events and setting *flags*. The flags used are listed in Table 4.

Depending upon symbol pairs (PS and NS, or PO and NS) the compiler calls into operation certain subroutines. These subroutines are listed in Table 6. Some of these are "generators" and produce commands in the temporary list or in the object program (see "ADD," for example), whereas others keep records or transfer data (TCNC).

8. Level

In order to control the priority of the various operations occurring in the formula, a level record, called L, is maintained. This may change as successive symbols of the formula are considered. The changes in L are given in Table 5 for consecutive pairs of operational symbols. The numbers in parentheses indicate the change in L which occurs *after* the generation of any commands for that pair of OP codes. Otherwise, the change in L occurs before the generation of commands.

Table 4
Flags

NAME	VALUES	MEANING
AC	0	The commands now in the object program are such that no partial result is in the accumulator.
Accumulator Flag	1	Accumulator contains partial result.
S	+2	Use "+" and "−" as is in formula.
Sign Flag	−2	Exchange "+" for "−" and "−" for "+".

Table 5
Change in Level for Pairs of OP Codes

PO \ NS	$\rightarrow$ p =	+ −	× /	([	)]
= p →	0	1	2	3	0
+ −	−1	0	1	(2)	0
× /	(−2)	(−1)	0	(1)	(−1)
)]	(−3)	−2	−1	0	−2
([	0	0	1	(2)	0

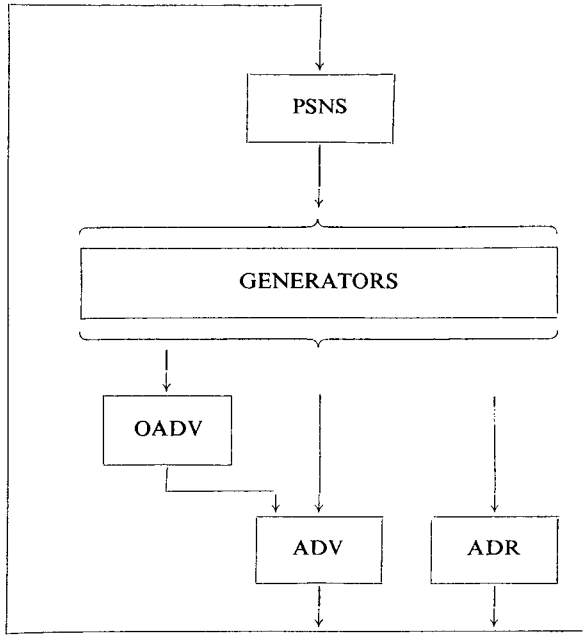

Fig. 1.—General structure of the Translator

9. General Structure of the Translator

The general structure of the translator is shown in Fig. 1. Depending upon PS and NS, or PO and NS, the PSNS distributor selects one of 81 generators (see Table 7). These generators make use of selected subroutines (Table 6) and via ADV (advance) or ADR (address) returns to PSNS for another distribution.

Certain pairs of symbols are called "illegal" and cause entry to the error routine (ERR). This routine causes a HALT, perhaps printing out PS and NS for diagnostic purposes.

In general, commands will be generated in the temporary list (TL) as long as L increases and the accumulator flag AC is "0". Whenever L decreases, commands are transferred from TL to OP until the level of commands in TL is less than the current level L. Each command in TL has stored with it the level (L), sign (S), and arithmetic working address (WA) in effect at the time that it was generated. The accumulator flag AC is "1" whenever execution of commands currently in the object program, OP, would leave a result in the accumulator. The flag S indicates a minus symbol which effects the sign of subsequent operations.

10. Examples

Two examples are given in Tables 8 and 9 to show how the translation takes place. The values of L, S, and WA are listed as they are during the generation of commands for that PO and NS. The details of address

Table 6

Subroutines

NAME	ROUTINE
Add	ADD: {A? {S < 0? "SU" → T, "AD" → T; CG(T).},,}.
Conditional Add	CD(T): {AC ≠ 0 ? CG(T) . CAD.}.
Address	ADR: NAM, PO → PS, PSNS.
Advance	ADV: NS → PS, SNS, PSNS.
Clear Add	CAD: {AC ≠ 0? ADD. {S < 0? "CS" → T, "CA" → T; CG(T).}}.
Clear Add—Store	CAST: PO = "→"? "ST" → T, {AC ≠ 0? "ST" → T, "CA" → T}, CG(T)}.
Command Generator	CG(T): {A + T → T, SF ≠ 0? {"AI" → U, TC(U), 0 → SF},, AC ≠ 0? NC, TC}.
Error	ERR: - - - -
Function	FUN: {A ≠ 0? FUNCTION DEFINITION? {"ST" + A → T, TC(T)}, {S < 0? {"NE" → T, TC(T)},; "DO" + A → T, TC(T),},,}.
Initialize	INIT: {0 → A → AC, 2 → S}.
Level	L(T): {L + T → L}.
Multiply—Inverse Divide	MUID: {AC ≠ 0? "MU" → T, "ID" → T, CG(T).}.
Name	NAM: - - - -
Next Command	NC(T): {T ≠ 0? {T → OP(CC), CC + 1 → CC},,}.
Operator Advance	OADV: 0 → A, NS → PO, ADV.
Present Symbol—Next Symbol	PSNS: 9 CL(PS) + CL(NS) + 1 → T, T.
Select Next Symbol	SNS: - - - -
Store	STORE: {"ST" → T, CG(T)}.
Temporary Store	TST(T): {AC ≠ 0? {"ST" + WA → U, NC(U), T + WA → T, TC(T), WA + 1 → WA, 0 → AC,},,}.
Temporary Command	TC(T): {T → TL(TI), L → TL(TI + 1), S → TL(TI + 2), WA → TL(TI + 3), TI + 4 → TI}.
Temporary Command—Next Command	TCNC: {TI − 4 → TI, −1 < TI? {L ≤ TL(TI + 3)? {TL(TI) → T, 1 → AC, NC(T), TCNC.}, TL(TI + 2) → S, TL(TI + 3) → WA;}, 2 → S; TI + 4 → TI.}.
Zero Advance	ZADV: 0 → T, TC(T), OADV.

Table 7

PS–NS Table

PS	L	P	(, [	),]	+, −	×	/	=, →
L	ERR. 1	ADR. 10	ADR. 19	ADR. 28	ADR. 37 46	ADR. 55	ADR. 64	ADR. 73
P	INIT, ADV. 2	ERR. 11	INIT, L(1), FUN, TC(0), L(2), OADV. 20	ERR. 29	L(1), ADD, ZADV. 38 47	L(2), CG(MU), ZADV. 56	L(2), CG(ID), ZADV. 65	NS = "→"? {CAD,TCNC}, STORE, OADV. 74
(, [	ADV. 3	ERR. 12	FUN, TC(0), L(2), OADV. 21	CAD, OADV. 30	ADD, OADV. 39 48	L(1), CG(MU), OADV. 57	L(1), CG(ID), OADV. 66	ERR. 75
),]	ERR. 4	L(−3), TCNC, 2 → S, OADV. 13	ERR. 22	L(−2), OADV. 31	L(−2), TCNC, OADV. 40 49	L(−1), TCNC, 2 → S, OADV. 58	L(−1), TCNC, 2 → S, OADV. 67	L(−3), TCNC, 2 → S, OADV. 76
+	ADV. 5	L(−1), CAD, TCNC, 2 → S, OADV. 14	TST(AD), FUN, TC(0), L(2), OADV. 23	CAD, OADV. 32	ADD, OADV. 41 50	L(1), TST(AD), CG(MU), OADV. 59	L(1), TST(AD), CG(ID), OADV. 68	L(−1), CAD, TCNC, 2 → S, OADV. 77
−	ADV. 6	L(−1), −S → S, CAD, TCNC, 2 → S, OADV. 15	TST(AD), −S → S, FUN, TC(0), L(2), OADV. 24	S → −S, CAD, OADV. 33	−S → S, ADD, −S → S, OADV. 42 51	L(1), −S → S, TST(AD), CG(MU), OADV. 60	L(1), −S → S, TST(AD), CG(ID), OADV. 69	L(−1), −S → S, CAD, TCNC, 2 → S, OADV. 78
×	ADV. 7	L(−2), CD(MU), TCNC, 2 → S, OADV. 16	TST(MU), FUN, TC(0), L(1), OADV. 25	CD(MU), L(−1), OADV. 34	L(−1), CD(MU), TCNC, OADV. 43 52	CG(MU), OADV. 61	MUID, OADV. 70	L(−2), CD(MU), TCNC, 2 → S, OADV. 79
/	ADV. 8	L(−2), CD(DI), TCNC, 2 → S, OADV. 17	TST(ID), FUN, TC(0), L(1), OADV. 26	CD(DI), L(−1), OADV. 35	L(−1), CD(DI), TCNC, OADV. 44 53	CD(DI), TCNC, 2 → S, OADV. 62	CD(DI), TCNC, 2 → S, OADV. 71	L(−2), CD(DI), TCNC, 2 → S, OADV. 80
=, →	ADV. 9	CAST, TCNC, 2 → S, OADV. 18	L(1), FUN, TC(0), L(2), OADV. 27	ERR. 36	L(1), ADD, ZADV. 45 54	L(2), CG(MU), ZADV. 63	L(2), CG(ID), ZADV. 72	STORE, OADV. 81

Table 8

Example 1

, $z = a - b \times (c \times (d + e \times f)/g \times (a - b) + h) \times k + c, \ldots$

PO	NS	L	S	AC	OP		TL					COMMANDS GENERATED BY	
					CC	COMMAND	TI	COMMAND	L	S	WA	GENERATOR	SUBROUTINES
,	=	0	2	0			0	ST *z*	0	2	0	74	STORE
=	−	1	2	0			1	AD *a*	1	2	0	45	ADD
−	×	2	−2	0			2	MU *b*	2	−2	0	60	CG(MU)
×	(	3	−2	0			3	0	3	−2	0	25	ZADV
(	×	4	−2	0			4	MU *c*	4	−2	0	57	CG(MU)
×	(	5	−2	0			5	0	3	−2	0	25	ZADV
(	+	5	−2	0			6	SU *d*	5	−2	0	39	ADD
+	×	6	−2	0			7	MU *e*	6	−2	0	59	CG(MU)
×	)	6	−2	0			8	CS *f*	6	−2	0	34	CAD
)	/	4	2	1	0 1 2 3	CS *f* MU *e* SU *d* MU *c*	3					67	TCNC
/	×	4	2	1	4	DI *g*						62	CD(DI)
×	(	4	2	0	5	ST 0	4 5	MU 0 0	4 5	2 2	0 1	25	TST(MU) ZADV
(	−	5	2	0			6	AD *a*	5	2	1	39	ADD
−	)	5	−2	0			7	CS *b*	5	−2	1	33	CAD
)	+	3	2	1	6 7 8	CS *b* AD *a* MU 0	2					40	TCNC
+	)	3	−2	1	9	SU *h*						32	CAD
)	×	2	2	1	10	MU *b*	2					58	TCNC
×	+	1	2	1	11 12	MU *k* AD *a*	1					43 43	CD(MU) TCNC
+	,	0	2	1	13 14	AD *c* ST *z*	0					14 14	CAD TCNC

generation are not shown; in fact, the variables *a*, *b*, etc., are left in the address positions, although, in practice, actual memory locations may occur here. Another way to state this is to point out that two PSNS distributions are not shown for each PO and PS. That is, there is a distribution on PS = "," and NS = "*z*", then one on PS = "*z*" and NS = "=", and the third (shown) on PS = PO = "," and NS = "=". On occasion zeros are entered in the command position of the temporary list in order to record the current sign (S). Some of these are redundant and could be eliminated with some complication in the logic.

11. Functions and Subscripted Variables

The use of the temporary list facilitates the processing of functions and subscripted variables. Distinguishing functions $f(x)$ from subscripted variables $a(i)$ is best done by entering characterizing information in the name list. However, brackets will be used to denote subscripts as in $a[i]$ and a subscript flag, SF, will control the compiling process. Example 2 shows an object computer program for a nested sequence of function and subscript situation. The definition of functions is not covered by the generators of Table 7.

12. Extensions

The only problem in extending these techniques to functions of several variables (procedures) is that of providing storage for the arguments. This may be done by allotting space in the object program immediately following the subroutine transfer (DO) command.

Note that the present system communicates with subroutines by supplying the actual variables instead of the names of variables, and that variables not in the list of arguments are "universal" (that is, they have the same meaning inside and outside the subroutine). Certainly, in a practical application, the algorithm should be extended so that subscripts are not universal.

Table 9

Example 2

$+f(a+b[i+f(x)])-$

PO	NS	L	S	AC	OP		TL					COMMANDS GENERATED BY	
					CC	COMMAND	TI	COMMAND	L	S	WA	GENERATOR	SUBROUTINES
+	(	1	2	1 0	1	ST WA		AD WA DO *f*	1 1			23	TST(AD) FUN
(	+	3						AD *a*	3			39	ADD
+	[	3						CA *b* AI 0	3 3			23	FUN
[	+	5						AD *i*	5			39	ADD
+	(	5						DO *f*	5			23	FUN
(	)	7						CA *x*	7			30	CAD
)	]	5										31	
]	)	3										31	
)	−	1		1	2 3 4 5 6 7 8 9	CA *x* DO *f* AD *i* AI 0 CA *b* AD *a* DO *f* AD WA						49	TCNC

13. Acknowledgements

Simple arithmetic compiling techniques are described in Huskey, Halstead and McArthur (1960), and alternate procedures are given in Samelson and Bauer (1960) and Backus *et al.* (1957).

Final organization of the material in this paper was done while the author was on sabbatical leave from the University of California at the Mathematical Centre in Amsterdam and at Cambridge University in England. Part of the work was supported by the Bendix Corporation. Rudimentary compilers to check the described algorithms were developed on the X-1 computer of N.V. Electrologica, and on EDSAC 2 at Cambridge. A final check was made on the Bendix G-15, and Mr. W. H. Wattenburg discovered and removed a couple of errors while checking the algorithm on the IBM 704 at the University of California.

References

HUSKEY, H. D., HALSTEAD, M. H., and MCARTHUR, R. (1960). "NELIAC—A Dialect of ALGOL," *Communications of the Assoc. for Computing Machinery*, Vol. 3, No. 8, p. 463.

SAMELSON, K., and BAUER, F. L. (1960). "Sequential Formula Translation," *Communications of the Assoc. for Computing Machinery*, Vol. 3, No. 2, p. 76.

BACKUS, J. W., *et al.* (1957). "The Fortran Automatic Coding System," *Proc. Western Joint Computer Conf.*, Los Angeles, p. 188.

IFIP CONGRESS 62

Call for Papers

The International Federation of Information Processing Societies (IFIPS) will hold a Congress in Munich, Germany, from 27 *August to* 1 *September* 1962.

The Congress will cover all aspects of Information Processing and Digital Computers including the following:

(1) *Business Information Processing*
e.g. data processing in commerce, industry, and administration.

(2) *Scientific Information Processing*
e.g. numerical analysis; calculations in applied mathematics, statistics, and engineering; data reduction; problems in operations research.

(3) *Real Time Information Processing*
e.g. reservation systems; computer control; traffic control; analog-digital conversion.

(4) *Storage and Retrieval of Information*
e.g. memory devices; library catalogues.

(5) *Language Translation and Linguistic Analysis*

(6) *Digital Communication*
e.g. encoding; decoding; error detecting and error correcting codes for digital data transmission.

(7) *Artificial Perception and Intelligence*
e.g. pattern recognition; biological models; machine learning; automata theory.

(8) *Advanced Computer Techniques*
e.g. logical design; logical elements; storage devices; ultra high-speed computers; program techniques; ALGOL.

(9) *Education*
e.g. selection and training of computer specialists; training of non-specialists in the use of computers; information processing as a University subject.

(10) *Miscellaneous Subjects*
e.g. growth of the information processing field.

In each category it is planned to cover, where appropriate, the applications of digital computers, programming, systems design, logical design, equipment, and components.

Those wishing to offer papers are invited to send abstracts of 500–1,000 words to:

M. V. Wilkes,
The British Computer Society,
c/o University Mathematical Laboratory,
Corn Exchange Street,
Cambridge,

by 15 *September* 1961. These abstracts will be considered by the international program committee of IFIPS, and authors of selected abstracts will be invited to submit their complete papers (in French or English) for consideration by the program committee in March 1962.

In addition to accepted papers, there will be invited papers, symposia, and panel discussions.