NSWC/WOL/TR 75-109

NSWC TECHNICAL REPORT

WHITE OAK LABORATORY

A COMPARATIVE DESCRIPTION OF SEVERAL HIGH LEVEL COMPUTER LANGUAGES

BY
C. Nicholas Pryor

9 JULY 1975

NAVAL SURFACE WEAPONS CENTER
WHITE OAK LABORATORY
SILVER SPRING, MARYLAND 20910

NAVAL SURFACE WEAPONS CENTER
WHITE OAK, SILVER SPRING, MARYLAND 20910

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NSWC/WOL/TR 75-109	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A COMPARATIVE DESCRIPTION OF SEVERAL HIGH LEVEL COMPUTER LANGUAGES		5. TYPE OF REPORT & PERIOD COVERED Final
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) C. Nicholas Pryor		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Surface Weapons Center White Oak Laboratory White Oak, Silver Spring, Maryland 20910		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 9 July 1975
		13. NUMBER OF PAGES 48
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Digital
Computer
Programming
Languages

PRICES SUBJECT TO CHANGE

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A number of high level computer languages are in current use or under development for military applications. These include FORTRAN, BASIC, ALGOL, PL/1, PL/M, CMS-2, JOVIAL, CS-4, and SPL-1. These languages are compared by comparing similar statement and declaration types in each language, and by an example program written in each of the languages. The basic facilities in each language are shown to be similar, with the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

differences mostly cosmetic in nature. A recommendation is made that the languages under development be derived from a common base, possibly a subset of the commercial language PL/1, to avoid unnecessary variety.

19. FORTRAN
BASIC
Algol
PL/1
PL/M
CMS-2
JOVIAL
CS-4
SPL-1

NSWC/WOL/TR 75-109 9 July 1975

PREFACE

This report describes and compares several high level computer languages in use or considered for military applications. The purpose is to provide in a single source a cursory overview of several languages, without distracting detail on any. Thus, it provides an introduction to these languages and their capabilities for those generally familiar with computer languages, but not with the specifics of the languages covered. While certain of the languages discussed are commercial products, no endorsement of these products or their parent companies is implied. This work was performed in support of overall DoD interest in military computer languages.

C. Nicholas Pryor

C. NICHOLAS PRYOR
By direction

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

A COMPARATIVE DESCRIPTION OF SEVERAL HIGH LEVEL COMPUTER LANGUAGES

Prepared by:

C. Nicholas Pryor

INTRODUCTION

1. There is considerable interest in the Department of Defense at the present time in establishing a common high level programming language for use across a wide range of computing applications. A number of languages currently exist and are in use for scientific or tactical computation. Some of these are proposed as candidates for a common language; the alternative is to define an entirely new language based on the best features from existing languages. Although each of the existing languages has some special features peculiar to itself, the basic computational and sequence control functions of all the languages are quite similar. The purpose of this report is to investigate the basic statement types that are common to all languages and to compare them on a side-by-side basis for several commonly used or proposed languages. The basic statements are found to be similar in functions among the languages studied, with differences that are more cosmetic than fundamental. There is no attempt to make judgments concerning the differences among the languages.

2. Three categories of languages are included. FORTRAN, BASIC, ALGOL, and PL/1 are existing commercial languages in wide application, with PL/M being a simplified offshoot of PL/1. CMS-2 and JOVIAL are currently standard Navy and Air Force tactical languages respectively. CS-4 and SPL-1 are new languages currently under development.

SOME CAVEATS

3. This report is intended to provide only a superficial level of understanding of the differences between the languages, generally at the level at which the unsophisticated programmer would be interested. Thus some subtleties of the differences are not discussed, and not all the options are shown. Absolute precision or consistency of notation between languages was not attempted

Preceding page blank

where this might obscure the basic similarity between the languages. Each language has certain statement types or capabilities which are not discussed here, although the few statement types discussed constitute the vast majority of all programs. Minimum emphasis is placed on data types accepted by the language, on declarations of variables or procedures, or on means of parameter passing to procedures. These subjects are worthy of a separate effort.

4. It is surprisingly hard to find a single concise definition of most languages. The older languages tend to have extensions added by individual implementers, or to have evolved into several distinct dialects under the same name. This is evidenced by the difficulties encountered in transporting a program written in "standard" FORTRAN from one machine to another. Newer languages still in development tend to change more rapidly than their documentation, so one can at best get a snapshot of their descriptions. Thus an attempt was made to check several sources where possible, to sort out the "standard" part of the language from the implementation-dependent features.

5. Much of the material for this report came from references (1) and (2) which contain brief histories and descriptions of many of the languages, circa 1967. More detailed material for each language was drawn from references (3) through (11). Where several dialects of a language exist, the information in this report was drawn primarily from the reference cited.

---

(1) Sammet, Jean E., Programming Languages: History and Fundamentals, Prentice Hall, 1969.
(2) Rosen, Saul, Programming Systems and Languages, McGraw-Hill, 1967.
(3) IBM 7090/7094 IBSYS Operating System; Version 13 FORTRAN IV Language. IBM Sys. Ref. Lib. Form C28-6390-0.
(4) Kemeny, J. G. and Kurtz, T.E.; A Manual for Basic; Dartmouth College, 1965 (printed by C-E-I-R Inc.).
(5) Nauer, Peter et al; Revised Report on the Algorithmic Language ALGOL 60; Communications of the ACM; Jan 1963.
(6) Bates, F. and Douglas, M.L., Programming Language/One, Prentice-Hall, 1967.
(7) A Guide to PL/M Programming, Intel Corporation, Sep 1973.
(8) Users Reference Manual for Compiler-Monitor System (CMS-2) for use with AN/UYK-7 Computer, Sperry-Univac, Oct 1973 revision (prepared for NAVSHIPSYSCOM).
(9) Univac 1100 Series EXEC 8 JOVIAL Programmer Reference Manual, Sperry-Univac Form UP-7698, undated.
(10) CS-4 Language Reference Manual and Operating System Interface, Intermetrics, Inc. Dec 1973.
(11) Cornyn, J. J. and Smith, W.R., Baseline Definition of a High-Level Real-Time Language for Digital Signal Processing (SPL/1), Naval Research Laboratory, 28 Feb 1975.

HISTORICAL SUMMARY OF LANGUAGES STUDIED

6. Development and widespread acceptance of high level languages for computers began with the proprietary development of FORTRAN by IBM in the 1950's, followed shortly by the international committee definition of ALGOL and the definition of the business language COBOL. Most of the succeeding commercial or military languages can be traced to these starts, and this ancestry is of some interest in understanding a language and its supporters. Figure 1 is an attempt to trace this ancestry for the languages studied here, and the following paragraphs provide some more information on each. Considerable additional information on the development of several of these languages can be found in reference (1).

7. Figure 1 arbitrarily divides the languages into three generations. The first generation languages were developed fairly independently in the 1950's, on a relatively small base of experience in high level languages. The second generation languages are mostly products of the 1960's or early 1970's and had opportunity to select the best of the existing language approaches. Improvements in run-time software environments also made these languages more usable in time-critical applications. The third generation languages currently under development take additional advantage of modern programming concepts such as structured programming and of advanced hardware designed specifically to support the high level language constructs.

FORTRAN

8. FORTRAN can be considered to be the first of the true high level languages. It was first described by IBM in 1954 and was available for use on the 704 computer by 1957. Since the language was developed specifically for the 704, its original form was somewhat machine dependent. These dependencies were largely eliminated with the development of FORTRAN IV in 1962, which was the first language to be documented as an ANSI standard. FORTRAN is available for virtually every production computer in the United States and is almost universally known by scientific computer programmers. Although FORTRAN has some faults that are widely condemmed by computer scientists, it is still firmly entrenched as the standard computer language in the U.S. This can be attributed to the large reservoir of FORTRAN programmers, widespread implementation with very efficient compilers, and the excellent I/O facilities built into the language. FORTRAN is also the language in which the compilers for many other languages are implemented.

(1) Sammet, Jean E., Programming Languages: History and Fundamentals, Prentice Hall, 1969.

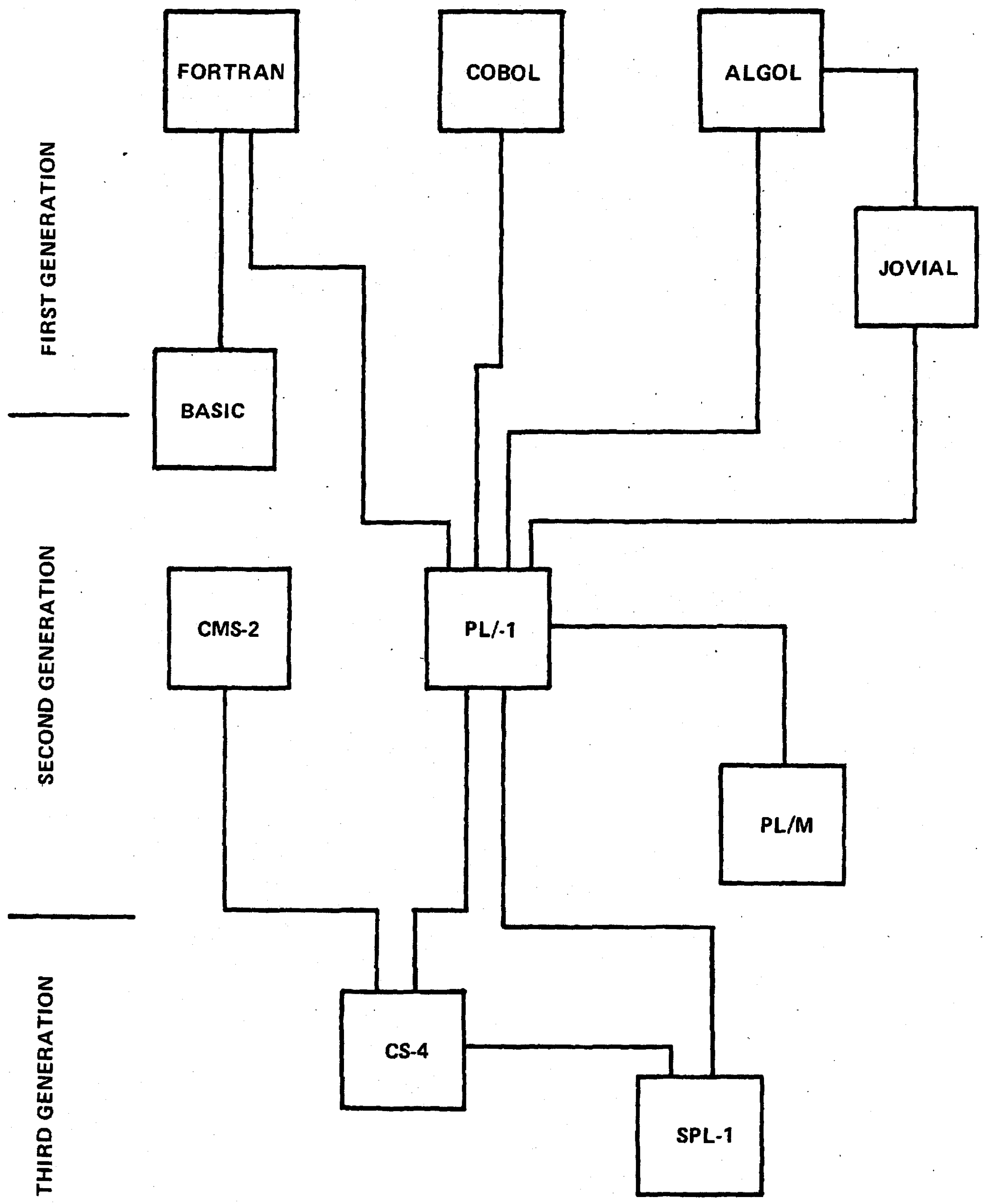

FIG. 1 ANCESTRY OF HIGH LEVEL LANGUAGES STUDIED

BASIC

10. The BASIC language is a simple language developed at Dartmouth College in 1965 and is designed primarily for interactive use from remote terminals. BASIC is a simplified, easy-to-teach, derivative of FORTRAN, although it is not a subset of FORTRAN. Currently, BASIC is available on most time-sharing computer systems and on many mini-computers. Because of its wide availability and ease of use, it is now used nearly as much as FORTRAN for small-scale scientific computing. While BASIC is not designed as a real-time control language, it is included here to show that it has many of the same basic facilities as the more powerful languages.

ALGOL

11. ALGOL is also one of the early high level languages, based on the work in 1958 of an international committee to define a standard computer language. Further refinement of the language in 1960 led to the definition of ALGOL 60, which is the version generally implemented. ALGOL has received widespread support in Europe, where it is nearly universally used in scientific computing. However, FORTRAN had acquired enough momentum in the U.S. that most manufacturers chose to implement FORTRAN compilers on their hardware rather than risk introduction of ALGOL. Even today, ALGOL is only minimally supported by most U.S. computer manufacturers.

PL/1

12. In 1963, IBM convened a committee of users to develop a "major advance in FORTRAN," to provide a general purpose language having the capabilities of FORTRAN, ALGOL, COBOL, and JOVIAL. The result was PL/1, introduced in 1966 along with the IBM 360 series computers. Although the language has been defined as an ANSI standard, it is still not widely implemented on other than IBM hardware. Originally it was considered that PL/1 would rapidly replace FORTRAN, but today after some ten years of availability it has made little progress. Long term prospects for PL/1 are still good however, since it is able to satisy both FORTRAN and ALGOL devotees and it contains most features considered important in a high level language. In fact, some complain that it is too capable, since run-time inefficiencies can result if all its facilities are implemented.

PL/M

13. PL/M is a simplified high level language developed in 1973 by the Intel Corporation to support its line of microprocessors. The language is nearly a subset of PL/1 and contains only those features necessary to support small process-control type systems. The purpose for including PL/M in this study is to illustrate the feasibility of using a subset of a more powerful language to develop a simple language with most of the same basic functions.

CMS-2

14. The CMS-2 programming language was developed by Univac beginning in about 1970, to be used in conjunction with the Navy standard UYK-7 computer. This language is currently the Navy standard high level language for tactical data systems and is, or soon will be, available for the standard UYK-7 and UYK-20 computers. While many programmers have raised objections to various features of CMS-2, its basic features are generally similar to other languages studied. Some of these objections may be not so much to the language itself, but to the implementation of the compiler and the run-time operating system and to the fact that its use was mandated before support software was fully available.

JOVIAL

15. The JOVIAL language is the result of effort at System Development Corportation in 1958 to develop a standard language for use in Air Force projects. JOVIAL was under development at the time of the initial ALGOL conference, and its definition was strongly influenced by this early ALGOL work. A JOVIAL compiler was operating by 1960, and JOVIAL became an Air Force standard language for command and control about 1967. JOVIAL is also supported to some extent on commercial computers.

CS-4

16. CS-4 is the major new language designed to accompany the introduction of the Navy "All Applications Digital Computer" in the early 1980's. It is intended primarily as a data processing language, and its features are designed to be compatible with the advanced hardware concepts of the AADC program. CS-4 supposedly contains a core language called METAPLEX, with extensions to provide the full data processing capabilities of CS-4. Other extensions of METAPLEX were intended to form various special purposes languages such as for system control and signal processing. Major development of CS-4 took place beginning in about 1972 under an NELC contract to Intermetrics. The language development is now in a state of limbo due to a major restructuring of the overall AADC program.

SPL-1

17. SPL-1 is a special purpose signal processing language, originally intended to be part of the CS-4 family of languages within the AADC program. This language was developed by NRL and Intermetrics, primarily during 1974, and is the high level language currently intended for the Navy Advanced Signal Processor program. Even though this language is for a special application and has special features for control of external processes, the basic internal functions of the language are quite similar to those of the general purpose languages. However, in spite of the original attempts to provide a common base for CS-4 and SPL-1, some minor differences have developed

COMPARISON OF BASIC LANGUAGE FEATURES

18. In order to display the similarities and differences among the languages studied, this section compares the handling of features common to the languages. The first subsections compare the basic program structure, labeling and separation of statements, and the data types recognized by each language. Then a comparison is made among languages for each of the following general statement types, which comprise most of the statements in a typical program:

Assignment statements

Unconditional Branch Statements (GO TO)

Conditional (two-way) Branch Statements (IF)

Multiple Case Branch Statements (CASE)

Loop Control Statements

Procedure Call Statements

Means of declaring variables and their attributes and forms for procedure declarations are also compared.

19. Table 1 summarizes some definitions and the notation used in this report for presenting the format of the various statement and declaration types.

TABLE 1. SUMMARY OF NOTATIONS USED

aexp	represents an arithmetic expression, which has a numerical value
attributes	attributes of a data variable including type, storage method, range, dimensions, etc.
dim	specification of dimensions of an array
exp	represents an expression which may be either arithmetic or logical
label	represents a label (alphanumeric or numeric depending on language) corresponding to that used to identify a statement elsewhere in the program
lexp	represents a logical expression, having the value true or false
name	represents the name of a subroutine or function procedure
num	represents a numeric constant
packing	an array data attribute determining the packing of data items into computer words
statement	represents a single program statement, or a compound statement or block which functions as a single statement
statement list	represents a sequence of one or more statements, where the length is not specified
storage	a data attribute indicating the type of storage (e.g., static or dynamic) used for a variable
swname	represents the name used to identify a switch declaration
type	a data attribute representing the data type, such as Integer, Real, or Logical
value	the initial value to which a variable is set
var	represents the name used to identify a variable
....	represents optional repetition of the same form
[ ]	represents an optional clause

Block Structure

20. Block structure in a computer program supports the desirable top-down approach to computer programming by allowing each functional segment of a program to be viewed as a block, perhaps containing several blocks of more detailed coding. If the program is partitioned properly, a programmer at each level only needs to know the function performed by a given block without needing to know the implementation within the block. Similarly, programming or modification within a block should not be apparent to other blocks in the program.

21. Block structure is accomplished in a language by defining that a block of statements enclosed by BEGIN...END (or equivalent) key words appears to the outside exactly like a single statement, and can be used in place of any statement. Thus each block internally functions as a simple list of statements, even though some of these "statements" may actually be blocks nested to several levels.

22. Sometimes it is necessary to define additional variables or to label some statements within a block. In order to make programming of blocks independent, these variable and label definitions are known only within the block and any other blocks it contains. They are not visible outside the block in which they are defined. Some languages distinguish between blocks and compound statements, where the difference is that no new variables may be declared in a compound statement. The compound statement is somewhat more efficient to execute at run time. The degree to which block structure is supported by each of the languages studied is listed here:

FORTRAN:	No block structure defined. However nested subroutine calls and common storage provide many equivalent features.
BASIC:	Block structure not supported.
ALGOL:	Blocks are compound statements enclosed by BEGIN....END structure.
PL/1:	Blocks and enclosed by BEGIN....END structure, while compound statements are enclosed by DO....END structure.
PL/M:	Blocks and compound statements are enclosed by DO....END structure
CMS-2:	Blocks and compound statements are enclosed by BEGIN....END structure.
JOVIAL:	Blocks and compound statements are enclosed by BEGIN....END structure.
CS-4:	Blocks and compound statements are enclosed by BEGIN....END structure.
SPL-1:	Blocks and compound statements are enclosed by BEGIN....END structure and must be labeled.

Separation of Statements

23. Each program is composed of a number of statements and declarations (non-executable statements) which must be separated in some way. The method used in each language to identify the end of one statement and the beginning of the next is summarized below:

FORTRAN:	One statement per card (input line) of essentially fixed format. End of card represents end of statement unless a a special continuation card symbol used.
BASIC:	One statement per input line, flexible format.
ALGOL:	Free format. Statement terminated by a semicolon. Multiple statements per line allowed.
PL/1:	Free format. Statement terminated by a semicolon. Multiple statements per line allowed.
PL/M:	Free format. Statement terminated by a semicolon. Multiple statements per line allowed.
CMS-2:	Free format. Statement terminated by a dollar sign. Multiple statements per line allowed.
JOVIAL:	Free format. Statement terminated by a dollar sign. Multiple statements per line allowed.
CS-4:	Free format. Statement terminated by a semicolon. Multiple statements per line allowed.
SPL-1:	Free format. Statement terminated by a semicolon. Multiple statements per line allowed.

All of the block structured languages (that is, excluding FORTRAN and BASIC) are thus similar in this respect except for the trivial substitution of the dollar sign for the semicolon in CMS-2 and JOVIAL.

Statement Labels

24. It is occasionally necessary in one part of a program to refer to another point in the program, as in indicating the destination of a GO TO or in calling a procedure. (Note this is different from referring to a variable.) Each language therefore has some means of labeling statements. In each case, the label comes before the remainder of the statement. The method and requirement for labeling is described below for each language studied:

FORTRAN:	Any statement may be preceded by a numeric label.
BASIC:	Every statement must be preceded by a numeric label.
ALGOL:	Any statement (or block) may be preceded by one or more alphanumeric labels, with each followed by a colon.
PL/1:	Any statement (or block) may be preceded by one or more alphanumeric labels, with each followed by a colon.
PL/M:	Any statement (or block) may be preceded by one or more alphanumeric labels, with each followed by a colon.
CMS-2:	Any statement (or block) may be preceded by an alphanumeric label, followed by a period.
JOVIAL:	Any statement (or block) may be preceded by an alphanumeric label having two to six characters followed by a period.
CS-4:	Any block may be preceded by an alphanumeric label, followed by a colon.
SPL-1:	Any statement may be preceded by an alphanumeric label, followed by a colon. Every block must have a label.

Data Types Supported

25. One of the significant differences among the languages is the variety of data types supported by each. These are summarized in Table 2. Nearly universal types are Logical (representing true or false), Fixed Point Integers, and Floating Point Real numbers. Additional types are supported by some languages for special purposes. These include double precision arithmetic for the scientific languages, complex fixed point arithmetic to support signal processing operations, and fixed point scaled (or mixed number) arithmetic for tactical languages with navigation and similar functions to perform. Most languages also support character string data types, useful for display and operator interaction. The entry under arrays shows the maximum number of array dimensions allowed by the language. Subtleties such as the form in which each data type is stored or the mixtures of types allowed in expressions are not covered here.

TABLE 2. DATA TYPES RECOGNIZED BY LANGUAGES

	FORTRAN	BASIC	ALGOL	PL/1	PL/M	CMS-2	JOVIAL	CS-4	SPL-1
Logical (Boolean or Bit)	x		x	x	x	x	x	x	x
Integer (Fixed Point)	x		x	x	x	x	x	x	x
Mixed Number (Fixed Point)				x		x	x		
Real (Floating Point)	x	x	x	x		x	x	x	x
Double Precision (Floating Point)	x			x					
Complex (Floating Point)	x			x				x	x
Complex (Fixed Point)				x					x
Character Strings			x	x	x	x	x	x	x
Arrays (number of dimensions)	3	3	n	32	1	7	10	n	*

* implementation dependent

n arbitrarily large

Data Declarations

26. Most languages require certain characteristics of each data variable to be specified through a non-executable Data Declaration, to provide information to the compiler on allocation of storage and type of arithmetic operations required. For single variables this information may be fairly simple, while for arrays it is relatively complex. FORTRAN and BASIC generally only require declaration of array variables, while the remaining languages require each variable name to appear in a declaration. Some languages allow data declaration to be intermixed with executable statements in the program, while others require all declarations in a given block to appear before the executable program segment. Data names declared within a block are defined only within that block or other blocks nested within it, and in some cases storage is allocated only while the block is executing.

27. Table 3 shows the basic form of the data declaration in each language. Generally a declaration contains the variable name (or a list of names) and certain attributes such as data type, range, storage, etc. In some languages the dimensions of array variables are included in the basic declaration, while in others a separate form of declaration is used for arrays. Most of the newer languages also provide a means in the declaration of setting the initial value of a variable. Details of the attributes that can be specified for variables in the various languages are beyond the scope of this report.

TABLE 3. FORMS OF VARIABLE AND ARRAY DECLARATION

FORTRAN:		[TYPE] type var,...
		DIMENSION var(dim),...
BASIC:		DIM var(dim),...
ALGOL:		[OWN] type var,...
		[OWN] [type] ARRAY var,...(dim),...
PL/1:		DECLARE var[(dim)] [attributes] [INITIAL(value)],...
PL/M:		DECLARE var [(dim)] type [INITIAL(value,...)]
	or	DECLARE (var [(dim),...)] type [INITIAL(value,...)]
CMS-2:		VRBL var type [P value]
	or	VRBL (var,...) type [P value]
		TABLE var storage packing [INDIRECT] dim
JOVIAL:		ITEM var attributes [value]
		ARRAY var dim attributes
CS-4:		VARIABLE var IS [ARRAY(dim)] type [:= value]
	or	VARIABLES var,... ARE [ARRAY(dim)] type [:= value]
SPL-1:		VAR[storage] type var,... [:=value],...
		VAR [storage] type ARRAY var,...(dim) [:= value]

Assignment Statement

28. The assignment is the basic statement used for actual manipulations of data. Generally its function is to set the value of some variable equal to the value obtained by evaluating an expression. The forms used for the basic assignment statement in the languages studied are shown in Table 4. With the exception of the verbal as opposed to algebraic form used in CMS-2, all languages use a generally similar form. In some BASIC implementations, the LET key word is optional. The major remaining difference is the use in some languages of the := assignment operator to distinguish it from the = used as a relational operator. However, this does not seem to be required by the syntax, since PL/1 and PL/M manage to use the = for both purposes.

29. Remaining differences not illustrated here are that some languages allow multiple assignments by naming several variables on the left sides, and each language tends to have its own rules for handling mixed-mode situations where left and right parts or components of expressions do not agree in mode.

TABLE 4. FORM OF ASSIGNMENT STATEMENT

FORTRAN:	var = exp
BASIC:	LET var = exp
ALGOL:	var := exp
PL/1:	var = exp
PL/M:	var = exp
CMS-2:	SET var TO exp
JOVIAL:	var = exp
CS-4:	var := exp
SPL-1:	var := exp

Expressions and Operators

30. In any of the languages studied, an expression consists of one or more variable names or constants combined through the use of various arithmetic, relational, or logical operators according to the usual rules of algebraic notation. While the definitions of expressions and order of precedence of operations are similar in all the languages, some differences occur in the notation used to represent the operators. Some of this is caused by differences in the assumed input medium. For example the character set in Fortran is limited to the 47 characters available on an IBM 026 keypunch, while Basic is able to use other characters available on an ASR-33 teletype.

31. The common operators can be divided into three types as follows. Arithmetic operators are those which combine two arithmetic elements to yield an arithmetic value. Relational operators are those which compare two arithmetic elements to yield a logical (true or false) value. Logical operators are those which combine two logical values to yield a logical result. Some languages tend to permit mixed mode expressions where algebraic and logical data elements can be substituted for each other in expressions, with the compiler making some conversion, while others do not. Table 5 lists the common operators along with the notation used in each of the languages under study. In some languages either of two forms may be used for some operators. In these cases both forms are given in Table 5, with one above the other.

TABLE 5. NOTATION USED FOR OPERATORS

ARITHMETIC OPERATORS	FORTRAN	BASIC	ALGOL*	PL/1	PL/M	CMS-2	JOVIAL	CS-4	SPL-1
Addition	+	+	+	+	+	+	+	+	+
Subtraction (negation)	-	-	-	-	-	-	-	-	-
Multiplication	*	*	X	*	*	*	*	*	*
Division	/	/	/	/	/	/	/	/	/
Exponentiation	**	↑ **	↑	**		**	**	**	**
RELATIONAL OPERATORS									
Equal	.EQ.	=	=	=	=	EQ	EQ	=	=
Not Equal	.NE.	>< <>	≠	¬= NE	<>	NOT	NQ	~=	#
Greater Than	.GT.	>	>	> GT	>	GT	GR	>	>
Greater or Equal	.GE.	=> >=	≥	>= GE	>=	GTEQ	GQ	>=	>=
Less Than	.LT.	<	<	< LT	<	LT	LS	<	<
Less or Equal	.LE.	=< <=	≤	<= LE	<=	LTEQ	LQ	<=	<=
LOGICAL OPERATORS									
NOT	.NOT. .N.		¬	¬ NOT	NOT	COMP	NOT	~	NOT
AND	.AND. .A.		∧	& AND	AND	AND	AND	&	AND
OR	.OR. .O.		∨	| OR	OR	OR	OR	|	OR
EXCLUSIVE OR					XOR	XOR			XCR

* ALGOL Reference Language. Actual compilers generally use simpler forms, which vary from compiler to compiler.

Unconditional Branch Statement (GO TO)

32. The unconditional branch statement makes it possible to modify the sequence of execution. It has been shown that if the language contains suitable conditional and loop control statements, use of the unconditional branch statement is unnecessary. In fact, some studies have shown that the difficulty encountered in debugging or updating programs increases directly with the proportion of GO TO statements in the program, because of the additional obscurity of program flow. This is one basis of the structured programming concept, which attempts to eliminate the use of the GO TO statement. Nevertheless, all languages studied contained a form of the unconditional branch statement, as shown in Table 6. Since many of the languages ignore blanks within key words, the distinction between the GOTO and GO TO forms is seldom important. Thus, the unconditional branch statement is essentially identical in all languages.

33. One difference not discussed here is the restrictions on use of the GO TO incorporated in some languages. In FORTRAN and BASIC the use of the GO TO is nearly unrestricted, while some other languages do not permit a GO TO to branch across block boundaries. Languages with a restricted GO TO generally include an EXIT statement which causes a jump to the end of the current block. This capability is necessary in pure structured programming.

TABLE 6. FORMS OF UNCONDITIONAL BRANCH STATEMENT

FORTRAN:		GO TO label
BASIC:		GOTO label
ALGOL:		GO TO label
PL/1:		GO TO label
	or	GOTO label
PL/M:		GO TO label
	or	GOTO label
CMS-2:		GOTO label
JOVIAL:		GOTO label
CS-4:		GO TO label
SPL-1:		GO TO label

Two-Way Conditional Branch Statement (IF)

34. The conditional statement is the basic means of controlling logical flow of the program, depending on data values. Some expression is formed and tested, and the result of this test is used to determine the succeeding operations to be formed. The forms of the conditional branch statement are shown in Table 7. FORTRAN contains both an arithmetic IF statement and a logical IF where execution of a statement occurs only if a logical expression is true. The other languages use only the logical form.

35. In the FORTRAN algebraic IF and in BASIC, control is exercised by branching to some other labeled statement in the program. This tends to inhibit well structured programming. The remaining languages conditionally execute another statement imbedded within the IF statement, and then proceed to the next statement in sequence. Most also permit an ELSE clause so that either of two statements may be executed, depending on the result of the logical test. JOVIAL also contains an IFEITH construction which permits one of several instructions to be executed, depending on the first logical test to be found true.

36. Among the block-structured languages, SPL-1 is unique in that each of the other languages assumes a single statement following the THEN or ELSE clause. Of course in each case a block may be used in place of the single statement, but it still functions as a single statement. SPL-1 expects a statement list after the THEN or ELSE, and thus requires an IEND after each IF statement to terminate the list. Essentially this means that the THEN and ELSE terms automatically open new blocks in SPL-1, even if only one statement is to be used. Since SPL-1 does not permit branches outside of a block, this convention prohibits the "IF lexp THEN GO TO label" form which is often seen in programs written in the other languages.

TABLE 7. FORMS OF CONDITIONAL BRANCH STATEMENT

FORTRAN:	IF (lexp) statement
or	IF (aexp) label_1, label_2, label_3
BASIC:	IF lexp THEN label
ALGOL:	IF lexp THEN statement [ELSE statement]
PL/1:	IF lexp THEN statement [; ELSE statement]
PL/M:	IF lexp THEN statement [;ELSE statement]
CMS-2:	IF lexp THEN statement [$ ELSE statement]
JOVIAL:	IF lexp $ statement
or	
	IFEITH lexp_1 \$ statement_1 \$
	ORIF lexp_2 \$ statement_2 \$
	....
	ORIF lexp_n \$ statement_n \$
	END
CS-4:	IF lexp THEN statement [ELSE statement]
SPL-1:	IF lexp THEN
	statement list
	[ELSE
	statement list]
	IEND

Multiple Case Branch Statement (CASE)

37. The multiple case branch statement allows one of several operations to be performed by a program, depending on the value of some computed expression. It is thus a multi-choice decision, as compared to the True-False decision of the conditional IF statements. This type of statement is of particular use in compilers, assemblers, and emulators and in operational programs where multiple-mode selection is desired. Table 8 shows the form used for the multi-way branch in each of the languages studied.

38. Three rather distinct approaches are found in Table 8. First is the type used in FORTRAN and BASIC, where a branch is performed to one of n labeled statements elsewhere in the program, depending on the numerical value of an integer variable or expression. The second form is seen in ALGOL, CMS-2, and JOVIAL. Each of these uses a SWITCH declaration to list the branch destinations in the program, and the branch is executed by a GO TO statement which provides the numerical value to select the branch point. PL/1 has no explicit multi-way branch statement, but does allow indexing of an indirect branch through a table of pointers. This is essentially equivalent to the ALGOL approach. None of these approaches satisfies structured programming concepts, since the sequential flow of the program is interrupted.

39. The third type of multi-way branch is found in PL/M, CS-4, and SPL-1. Here the value of the expression computed by the CASE statement determines which one of the n statements immediately following the CASE statement is executed. After execution of this statement, control goes to the statement following the END, thus preserving the sequential flow of the program. Of course each statement between CASE and END may actually be a block, so no restriction is placed on the complexity of each operation. This form of the multi-way branch is now preferred because of its support of structured programming. The optional labels in CS-4 allow execution of a statement whose label matches a string expression, providing a powerful means of writing compilers or interpreters.

40. We have not discussed here the differences in the way each language handles out-of-bounds cases, where the value of the computed expression exceeds the length of the list of choices. We have also only discussed the index switch form of CMS-2 and JOVIAL. Each also has an item switch form with slightly different capabilities. A newer version of JOVIAL (J73) apparently has an executable SWITCH statement which is similar to the CASE statement in the other languages.

TABLE 8. FORMS OF MULTIPLE CASE BRANCH STATEMENT

FORTRAN:	GO TO (label_1,.....,label_n), var
BASIC:	ON aexp GOTO label_1, ...,label_n
ALGOL:	SWITCH swname label_1,....,label_n (non-executable declaration)
	.....
	GO TO swname (aexp)
PL/1:	no explicit facility
PL/M:	DO CASE aexp
	statement 1
	.....
	statement n
	END
CMS-2:	SWITCH swname label_1,...,label_n (non-executable declaration)
	....
	GOTO swname aexp INVALID label
JOVIAL	SWITCH swname=(label_1,...,label_n) (non-executable declaration)
	....
	GOTO swname ($aexp$)
CS-4:	CASE exp
	[label_1:] statement 1
	....
	[label_n:] statement n
	[OUTOFBOUNDS: statement]
	END
SPL-1:	DO CASE aexp OF
	statement 1
	....
	statement n
	[CELSE statement]
	CEND

Loop Control Statements

41. It is common in programming to have a group of statements which is to be repeated a number of times or until some test is satisfied. This leads to a construction known as a loop, in which a test is performed on each execution to determine whether to branch back and repeat the loop or to branch out and proceed to the next step of the program. To avoid the explicit need to write out the increment, test, and branch operations, each language has developed a compact means of specifying such loops. These loop control statements are shown in Table 9. Generally each language provides a means of indexing a variable on each pass through the loop from some initial value to some final value, with the option of setting the step to some value other than one. Additionally, some languages provide a WHILE test which causes the loop to repeat as long as the tested expression is true.

42. The first thing to notice from Table 9 is the lack of consistency even of the key word used to identify a loop-control statement. DO, FOR, VARY, and REPEAT are all used. There are also three methods used to identify the segment of code to be repeated within the loop. FORTRAN requires a specific label reference in the DO statement, identifying the final statement number to be included in the loop. This has been found to be a rather error-prone technique.

43. ALGOL and CS-4 assume the loop consists of a single statement to be repeated, and include this statement within the loop control statement. Of course a block can be substituted for the single statement, so there is no real restriction. The remaining languages recognize that in most cases several statements are required within the loop, so the loop control statement automatically opens a pseudo-block which must be terminated by an END statement. (The NEXT statement of BASIC is nearly equivalent to the END in the other languages.) The languages vary in whether they allow branches into or out of the statement list comprising the loop.

TABLE 9. FORM OF LOOP CONTROL STATEMENTS

FORTRAN:		DO label var = num_1, num_2[,num_3]
BASIC:		FOR var = $aexp_1$ TO $aexp_2$ [STEP $aexp_3$]
		....
		NEXT var
ALGOL:		FOR var:= $aexp_1$ STEP $aexp_2$ UNTIL $aexp_3$ DO statement
	or	FOR var :=exp_1 WHILE lexp DO statement
PL/1:		DO var =$aexp_1$ TO $aexp_2$ [BY $aexp_3$]
		statement list
		END
	or	
		DO WHILE(lexp)
		statement list
		END
PL/M:		DO var =$aexp_1$ TO $aexp_2$ [BY $aexp_3$]
		statement list
		END
	or	
		DO WHILE lexp
		statement list
		END
CMS-2:		VARY var [FROM $aexp_1$] THRU $aexp_2$ [BY $aexp_3$]
		statement list
		END
JOVIAL:		FOR var = $aexp_1$, $aexp_2$, $aexp_3$ \$
		BEGIN
		statement list
		END
CS-4:		[FOR var][FROM $aexp_1$][TO $aexp_2$][BY $aexp_3$] REPEAT statement
	or	WHILE lexp REPEAT statement
SPL-1:		FOR var FROM $aexp_1$ TO $aexp_2$ [BY $aexp_3$] DO
		statement list
		FEND

Procedure Calls

44. It is often desirable to write a single program segment which can be used by several other parts of the program. These common program segments are called procedures. It is necessary to provide a means of branching to the desired procedure and then returning to the calling point in the program when the procedure is completed. Since different parts of the program may want the same computation performed on different data, it is also necessary to provide a list of parameters to the procedure when it is called.

45. Each language recognizes two basic types of procedure, a function procedure and a subroutine. Functions are defined as those procedures returning a single value to the calling program. Examples are square root, trig functions, and the maximum value in a list of data. As seen in Table 10, each language permits a function procedure call by a simple reference to its procedure name followed by a parameter list in parentheses. This call may be imbedded in an expression, and the value of the function returned is equivalent to any other named variable. Some languages permit the function procedure to modify input parameters passed from the calling program; others do not.

46. Forms used for calling subroutines are shown in Table 11. In some languages the procedure name alone, followed by a parameter list if needed, is used as a complete statement. In others, a CALL or GOSUB key word is required. Subroutines generally perform some operation on some input parameters and modify the values of the same or other output parameters. In some languages no distinction is made between input and output parameters. In others, input and output parameters are explicitly separated. This provides some protection to the input parameters, since the subroutine is then not allowed to change them.

47. We have not distinguished the methods used in various languages for passing parameters between the calling program and the procedure during execution. Three basic methods are used and are termed "call by location," "call by value", and "call by name" respectively. Call by location" is the least flexible but the most efficient in execution, while "Call by name" is the most flexible but the least efficient to execute. Some languages permit the programmer to specify the means of parameter passing desired. However, in most cases, he has no control over, and little interest in, the method used until it creates some strange side effect in his program. The method is also compiler dependent in some languages.

TABLE 10. FORMS OF FUNCTION PROCEDURE CALL

FORTRAN:	name(parlist)
BASIC:	name(parlist)
ALGOL:	name(parlist)
PL/1:	name(parlist)
PL/M:	name(parlist)
CMS-2:	name(parlist)
JOVIAL:	name(parlist)
CS-4:	name(parlist)
SPL-1:	name(parlist)

TABLE 11. FORMS FOR SUBROUTINE PROCEDURE CALL

FORTRAN:	CALL name(parlist)
BASIC	GOSUB label
ALGOL:	name(parlist)
PL/1:	CALL name(parlist)
PL/M:	CALL name(parlist)
CMS-2:	name INPUT parlist OUTPUT parlist
JOVIAL:	name(input parlist = output parlist)
CS-4:	name(parlist)
SPL-1:	name(parlist)

Subroutine Procedure Declaration

48. The operations to be performed by a subroutine or function when it is called from another program are defined in a subprogram or procedure declaration. This declaration must contain the identity of the procedure, specification of a set of formal parameters corresponding to the actual parameters sent from the calling program, the list of operations to be performed, and a means of returning to the calling program.

49. The formats of the subroutine procedure declarations are shown in Table 12 for various languages. With the exception of BASIC where no parameter passing is possible, the first line in each case names the procedure and contains a list of the formal parameters. For each parameter passed to the subroutine, some means must be provided to identify the parameter type or other attributes and the means of passing from the main program. In some languages this information is contained within the parameter list itself, while in others it is contained in data declaration statements within the body of the subroutine. This can become a very complex subject when several means of parameter passing are provided and is beyond the scope of this report.

50. The statement list specifies the operations to be performed by the subroutine and may contain additional data declarations for variables to be used solely within the subroutine. Note that ALGOL assumes only a single statement in the procedure; this normally would take the form of a complex block. Finally the RETURN statement is used to cause a return to the calling program. Reaching the END statement of the procedure in most cases is equivalent to executing a RETURN.

TABLE 12. FORMAT OF SUBROUTINE PROCEDURE DECLARATION

FORTRAN:	SUBROUTINE name(parlist) statement list END
BASIC:	statement list RETURN
ALGOL:	PROCEDURE name(parlist); statement

PL/1:	name: PROCEDURE(parlist)
	statement list
	RETURN
	END name

PL/M:	name: PROCEDURE(parlist)
	statement list
	RETURN
	END name

CMS-2:	PROCEDURE name INPUT parlist OUTPUT parlist
	statement list
	RETURN
	END-PROC name

JOVIAL:	PROC name (input parlist = output parlist) BEGIN statement list [RETURN] END
CS-4:	name: proc-type(parlist) statement list END
SPL-1:	SUBROUTINE name(parlist) statement list END name

Function Procedure Declaration

51. Function procedures are similar to subroutines, except for their need to identify a single value to be returned to the calling program. Table 13 shows the form used for the Function declaration in each language. Generally the type of the returned variable is identified in the first line along with the function name. In some languages the value to be returned is determined by an expression in the RETURN statement. In others, it is produced by an appearance of the function name on the left side of an assignment statement within the procedure body.

52. BASIC has no true function procedure, just as it has no true subroutine facility. However, BASIC and FORTRAN both provide for a one-line function statement which equates a function name to a single expression.

TABLE 13. FORMAT OF FUNCTION PROCEDURE DECLARATION

```
FORTRAN:            name(parlist) = exp           (function statement)

              or    [type] FUNCTION name(parlist)
                    statement list                (external function)
                    END

BASIC:              DEF FNname (var) = exp

ALGOL:              type PROCEDURE name(parlist); statement

PL/1:               name:  PROCEDURE(parlist) [attributes]
                    statement list
                    RETURN (exp)
                    END name

PL/M:               name:  PROCEDURE(parlist) type
                    statement list
                    RETURN exp
                    END name

CMS-2:              FUNCTION name(parlist) type
                    statement list
                    RETURN (exp)
                    END-FUNCTION name

JOVIAL:             PROC name(parlist)
                    BEGIN
                    statement list
                    RETURN
                    END

CS-4:               name:  FUNCTION(parlist)
                    statement list
                    END

SPL-1:              FUNCTION name(parlist) type
                    statement list
                    RETURN exp
                    END name
```

UNIQUE ADDITIONAL FEATURES

53. In addition to the common data and statement types discussed in this report, each of the languages studied has certain additional features (or deficiencies) unique to the application for which it was designed. These may be considered to be special purpose extensions of a basic language, but they are often the reason for selection of one language or another. Some of the special features of the languages studied are listed below:

FORTRAN:	Very powerful formatted input/output provisions.
BASIC:	Interpretive execution. Built-in matrix operations.
ALGOL:	No input/output provisions in basic language.
PL/1:	Attempt at all-purpose language. Combines FORTRAN input/output capability with COBOL string-handling ability.
PL/M:	Designed as minimum high-level language for use with microprocessors. Derived from PL/1.
CMS-2:	Built-in data packing and shifting operations. Capability for inserting direct machine code.
JOVIAL:	Extensive means of defining complex tabular data formats.
CS-4:	Designed as an extensible language based on a core language METAPLEX.
SPL-1:	Provides for control and monitoring of external processes in multi-processor environment.

AN EXAMPLE

54. Further insight into the similarities and differences among the languages may be gained by comparing sample programs written in each language. The example chosen here is a sorting program which sorts a table of data into ascending order. The algorithm chosen is the "interchange sort" technique in which each adjacent pair of items in the table is compared, and the pair is interchanged if they are in the wrong order. The entire table is scanned in this way as many times as is necessary to produce the correct order throughout.

55. The necessary structure for the program is thus a loop which scans the table, comparing the value of each entry with that of its neighbor. If they are in the wrong order, the two entries are interchanged and a flag is set to indicate that a pair was swapped. After each pass through the loop, this flag is sensed to determine whether another pass is required. If the flag is found set, it is reset and the loop is repeated. If the flag is found reset, the sort operation is completed.

56. Table 14 shows the resulting programs in the nine languages considered. In each case the program is shown as a block of code which could be inserted in-line (that is, as an open subroutine) in a larger program. Thus, where appropriate, data declarations are included for those variables (I, SWAP, and TEMP) which are used internally by the sort algorithm. Although the flag SWAP could have been defined as a logical variable in most of the languages, it was treated as an integer in all cases. Often an algorithm such as this would be written as a closed subroutine or procedure to be called from another part of the program. In this case, a procedure declaration would have to be added to each of the programs shown. This procedure declaration must include a means of passing the location of the DATA array and its length N as parameters. The various means of doing this in the languages studied are considered beyond the scope of this report.

57. Although the algorithm is the same in each case, three different forms of program resulted. In the FORTRAN and BASIC programs, there is a single loop control statement used to scan the table on each pass. Within this loop is a logical IF statement used,if the data words are already in the right order, to jump around the statements which interchange the data words and set the swap flag. Outside the loop is another logical IF statement which tests the flag and causes a jump back to the loop if another pass is required.

58. The second form of program occurs with CMS-2, JOVIAL, and SPL-1, which are block structured languages but do not have a DO WHILE construct. In these languages the innermost IF statement contains a four-statement block which is executed only if the test is satisfied. At the end of the loop, a second IF statement is used to cause a jump to the beginning of the loop if another pass is

required. This turns out to be an awkward problem in SPL-1, since the THEN clause of an IF statement is not permitted to contain a GO TO which branches outside the THEN clause. The solution finally adopted was to enclose the entire sort program in a block and use an EXIT statement to get out if no further passes through the loop were required. This seems to be a problem that will occur frequently with SPL-1 as now defined, forcing the programmer to find ways to defeat the GO TO rules.

59. The third group of programs appears in the ALGOL, PL/1, PL/M, and CS-4 languages which contain a DO WHILE construct. While the program could have been written in each of these languages in the same form that was used in the second group (or even the first group), the DO WHILE was used to produce a pure structured (GO-TO-less) program. Here the loop used to scan the table is contained within a DO WHILE loop, which continues to execute as long as SWAP is set during each pass through the table. Since the WHILE test is evaluated at the beginning of the loop rather than at the end, SWAP must be set before entering the loop the first time. This is accomplished through the initialization option of the data declaration. Note that in these languages the entire executable program is actually contained within the DO WHILE statement structure.

60. No conclusions should be drawn from the slight differences in length among the various programs. The block structured programs were expanded and indented as much as possible in Table 14 to give the reader the best view of the logical structure of the program. Normally the programmer would compress the program somewhat, at least to the point of putting BEGIN and END brackets in line with other statements. As an example of (perhaps excessive) compression, the PL/M program could have been written

```
    DO; DECLARE (I,T) ADDRESS; DECLARE SWAP BYTE INITIAL(1);
    DO WHILE SWAP=1; SWAP=0; DO I=1 TO N-1;
    IF DATA(I) >DATA(I+1) THEN DO;
    T=DATA(I); DATA(I)=DATA(I+1); DATA(I+1)=T; SWAP=1;
    END; END; END; END;
```

Another misimpression that could be created by this example is related to the fact that ALGOL and CS-4 expect only a single statement following a loop control statement or an IF...THEN construct, while SPL-1 automatically opens blocks in both cases. PL/1 and PL/M assume single statements after an IF...THEN and open blocks after DO statements, since these are the most frequent requirements. However it happens in this example that the inner loop contains only the single IF statement, while the THEN clause of the IF statement contains a four-statement block. Thus both of the PL/1 assumptions were violated, resulting in some additional DO or END brackets.

TABLE 14. EXAMPLE PROGRAMS FOR INTERCHANGE SORT

TABLE 14a. FORTRAN PROGRAM FOR INTERCHANGE SORT

```
      INTEGER I, SWAP
      REAL TEMP
10    SWAP = 0
      DO 20 I = 1,N-1
      IF (DATA(I).LE.DATA(I+1)) GO TO 20
      TEMP = DATA(I)
      DATA(I) = DATA(I+1)
      DATA(I+1)  = TEMP
      SWAP=1
20    CONTINUE
      IF (SWAP.EQ.1) GO TO 10
```

TABLE 14b. BASIC PROGRAM FOR INTERCHANGE SORT

```
10 LET S = 0
20 FOR I = 1 TO N-1
30   IF D(I)< = D(I+1) THEN 80
40   LET T = D(I)
50   LET D(I) = D(I+1)
60   LET D(I+1) = T
70   LET S = 1
80 NEXT I
90 IF S = 1 THEN 10
```

TABLE 14c. ALGOL PROGRAM FOR INTERCHANGE SORT

```
BEGIN
  INTEGER I, SWAP;
  REAL TEMP;
  SWAP := 1;
  FOR I := 1 WHILE SWAP = 1 DO
    BEGIN
      SWAP :=0;
      FOR I := 1 STEP 1 UNTIL N-1 DO
        IF DATA[I]>DATA[I+1] THEN
          BEGIN
            TEMP := DATA[I];
            DATA[I] := DATA[I+1];
            DATA[I+1] := TEMP;
            SWAP := 1;
          END;
    END;
END;
```

TABLE 14. EXAMPLE PROGRAMS FOR INTERCHANGE SORT (continued)

TABLE 14d. PL/1 PROGRAM FOR INTERCHANGE SORT

```
BEGIN;
  DECLARE I FIXED, SWAP FIXED INITIAL(1);
  DECLARE TEMP FLOAT;
  DO WHILE (SWAP = 1);
    SWAP = 0;
    DO I = 1 TO N-1;
      IF DATA (I) > DATA(I+1) THEN
        DO;
          TEMP = DATA(I);
          DATA(I) = DATA(I+1);
          DATA(I+1) = TEMP;
          SWAP = 1;
        END;
    END;
  END;
END;
```

TABLE 14e. PL/M PROGRAM FOR INTERCHANGE SORT

```
DO;
  DECLARE (I,TEMP) ADDRESS;
  DECLARE SWAP BYTE INITIAL (1);
  DO WHILE SWAP = 1;
    SWAP = 0;
    DO I = 1 TO N-1;
      IF DATA(I) > DATA(I+1) THEN
        DO;
          TEMP = DATA(I);
          DATA(I) = DATA(I+1);
          DATA(I+1) = TEMP;
          SWAP = 1;
        END;
    END;
  END;
END;
```

TABLE 14. EXAMPLE PROGRAMS FOR INTERCHANGE SORT (continued)

TABLE 14f. CMS-2 PROGRAM FOR INTERCHANGE SORT

```
BEGIN $
  VRBL (I SWAP) I 16 U $
  VRBL TEMP F $
  LOOP.  SET SWAP TO 0 $
         VARY I FROM 1 THRU N-1 $
           IF DATA(I) GT DATA(I+1)  THEN
             BEGIN $
               SET TEMP TO DATA(I) $
               SET DATA(I) TO DATA(I+1) $
               SET DATA(I+1) TO TEMP $
               SET SWAP TO 1 $
             END $
         END $
         IF SWAP EQ 1 THEN
           GOTO LOOP $
END $
```

TABLE 14g. JOVIAL PROGRAM FOR INTERCHANGE SORT

```
BEGIN
  ITEM II I 16 U $
  ITEM SWAP I 16 U $
  ITEM TEMP F $
  LOOP. SWAP = 0 $
        FOR II = 1,1,NN-1 $
          BEGIN
            IF DATA(II) GR DATA(II+1) $
              BEGIN
                TEMP = DATA(II) $
                DATA(II) = DATA(II+1) $
                DATA(II+1) = TEMP $
                SWAP = 1 $
              END
          END
        IF SWAP EQ 1 $
          GOTO LOOP $
END
```

TABLE 14. EXAMPLE PROGRAMS FOR INTERCHANGE SORT (continued)

TABLE 14h. CS-4 PROGRAM FOR INTERCHANGE SORT

```
BEGIN;
  VARIABLES I,SWAP ARE INTEGER := 1;
  VARIABLE TEMP IS REAL;
  WHILE SWAP = 1 REPEAT
    BEGIN;
      SWAP := 0;
      FOR I FROM 1 TO N-1 REPEAT
        IF DATA(I) > DATA(I+1) THEN
          BEGIN;
            TEMP := DATA(I);
            DATA(I) := DATA(I+1);
            DATA(I+1):= TEMP;
            SWAP := 1;
          END;
    END;
END;
```

TABLE 14i. SPL-1 PROGRAM FOR INTERCHANGE SORT

```
SORT:  BEGIN
         VAR INT I,SWAP;
         VAR FLOAT TEMP;
         LOOP:  SWAP  := 0;
                FOR I FROM 1 TO N-1 DO
                  IF DATA(I) > DATA(I+1) THEN
                    TEMP := DATA(I);
                    DATA(I) := DATA(I+1);
                    DATA(I+1) := TEMP;
                    SWAP := 1;
                  IEND;
                FEND;
                IF SWAP = 0 THEN
                  EXIT;
                IEND;
                GO TO LOOP;
       END SORT;
```

SUMMARY AND CONCLUSIONS

61. As can be seen from the comparison of statement types and the example programs, the languages studied are not fundamentally different and offer quite similar capability, to the level of detail considered in this report. This is particularly true of the newer block structured languages, PL/1, PL/M, CS-4, and SPL-1. Since the superficial differences in format do not change the basic functions performed by each of the basic statement types, there seems to be no valid reason for lack of commonality at this level.

62. Since the newer languages, CS-4, SPL-1, and perhaps JOVIAL (J73) are not completely frozen, it would seem prudent for them to adopt a common form for the basic statement and declaration types. This could serve as the beginning of a truly common military language. Since PL/1 is a generally accepted existing commercial language having similar basic capabilities to the proposed military languages, this language provides an obvious model from which to derive a common language. This is not to suggest that all features of PL/1 are desirable in a common military language, as a full implementation may reduce efficiency in the portion actually needed. Rather a subset of PL/1 should be defined and extended as needed for military applications. PL/M may in this respect serve as a good starting point, since it essentially represents a minimum subset of PL/1.

63. Given that a military language is to be derived by appropriate extension of a PL/1 subset, two distinct approaches are possible. The first is to define a common subset, and to allow several special purpose languages (such as a signal processing language, a command and control language, and a scientific language) to be generated as separate extensionsof the basic language. This is essentially the approach begun with METAPLEX, from which CS-4 and SPL-1 were to be derived. The advantage is that each language is reasonably simple and efficient for the application at hand. Room is also provided for new specialized languages within the family or upgrading one extension without affecting the others if the need occurs. The other approach is to define one huge "umbrella" language, embodying all the facilities required by all application categories. This is essentially the approach originally taken by PL/1 in the commercial field. It has the advantage of allowing the programmer to mix the capabilities of several extensions where desired and assuring him that one compiler can handle anything he writes. However it has the effect of freezing the entire language, and it may actually become too complex to teach fully or to compile efficiently.