ESD-TR-72-330

MTR-2429

A GENERAL BASIS FOR COMPARATIVE EVALUATION OF AED, COBOL, JOVIAL, AND PL/1

by

J. C. Des Roches

FEBRUARY 1973

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 572B

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-71-C-0002

ESD-TR-72-330

MTR-2429

A GENERAL BASIS FOR COMPARATIVE EVALUATION OF AED, COBOL, JOVIAL, AND PL/1

by

J. C. Des Roches

FEBRUARY 1973

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 572B

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts

Contract No. F19628-71-C-0002

FOREWORD

The work described in this report was carried out under the sponsorship of the Deputy for Command and Management Systems, Project 572B, by The MITRE Corporation, Bedford, Massachusetts, under Contract No. F19628-71-C-0002.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved.

MELVIN B. EMMONS, Colonel, USAF
Director, Information Systems Technology
Deputy for Command & Management Systems

ABSTRACT

This report provides an analysis of the technical features and pertinent characteristics of the programming languages AED, COBOL, JOVIAL, and PL/I, which were chosen for evaluation because of general applicability to programming problems within the scope of Air Force interest. The methodology derives from the development of a Language Feature Outline and a Language Evaluation Questionnaire for which programmer/analysts supplied detailed technical information and subjective evaluations. The intent of this report is to provide material in support of evaluations of the relative suitability of the four languages for specific applications.

ACKNOWLEDGEMENT

The author acknowledges, with sincerity, that the contributions of the participants in the study group constitute, by far, the major content of the report and provide the substantive basis to which evaluation criteria may be applied by potential users of the subject languages. The staff members who comprised the study group and the assigned language responsibilities were as follows:

Mr. William Amory	JOVIAL
Mr. W. Reid Gerhart	PL/I
Mrs. Verniece Hensey	COBOL
Mr. Joseph E. Sullivan	AED

Others have contributed to lesser degrees and cannot all be thanked individually; Mr. John Glore, however, deserves mention for his contribution to JOVIAL material.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONC.)

LIST OF EXHIBITS

LIST OF TABLES

SECTION I

PURPOSE AND SCOPE

The purpose of this language comparison study is to provide an in-depth analysis of the technical features and pertinent characteristics of the programming languages AED, COBOL, JOVIAL, and PL/I. This analysis may then be used to aid in the evaluation of the applicability of each of the subject languages to different classes of programming problems within the scope of the Air Force interest, which is predominantly in the areas of data management, systems programming, and business applications.

Since most current procedure-oriented programming languages were designed to provide capabilities specific to a particular application area, large installations usually require a comprehensive set of languages in order to fulfill their diverse programming responsibilities in a timely and efficient manner. Recent history indicates that software cost has become the dominant factor in the total cost of automation, increasing substantially faster than hardware costs. As a result, it becomes mandatory that languages be effectively evaluated in terms of capabilities and limitations, so that the optimum match of programming language and application may be effected.

The subject languages, i.e., AED, COBOL, JOVIAL, and PL/I, were chosen because they provide advanced programming capabilities and versatility in their range of applicability. All have been in existence sufficiently long to have been subjected to the 'test of time'.

AED (Automated Engineering Design), developed at MIT's Electronic Systems Laboratory during the period 1959-1969 under Air Force

sponsorship and made public in 1969 by SofTech, Inc., was included since it supports a software engineering discipline and is particularly well suited to systems programming.

The development of COBOL (Common Business Oriented Language) traces back to 1959 when it was defined by an inter-company committee (CODASYL) under Department of Defense sponsorship. Its purpose was to provide a natural, English-like language suitable for business data processing with emphasis on machine independence. It has gained wide acceptance and become a USASI standard.

JOVIAL was developed about 1960 by the System Development Corporation under an Air Force contract. Its purpose was to combine numerical scientific facilities with a strong data management capability and thereby provide the features necessary for large scale information processing systems. The J-3 version of JOVIAL has become the Air Force standard programming language for Command and Control Systems.

PL/I, developed in 1963 by IBM and SHARE, represents a synthesis of features from FORTRAN, ALGOL, and COBOL and was specifically designed to support the functions of third-generation computer systems. It is a very powerful language, capable of scientific, business, and systems programming and, as such, offers perhaps the best potential for a 'one-language' installation.

This report is addressed to readers with a knowledge of higher-level programming languages and concepts. Its purpose is to delineate information relative to the salient features of the subject languages, rather than to provide a complete, rigorous language specification. To this end, emphasis has been placed on <u>what</u> is included in each language rather than <u>how</u> it is implemented.

SECTION II

METHODOLOGY

Language descriptions and evaluations may take many forms, primarily dictated by the purpose of the exposition; for example, language designers and implementors require formal and rigorous language specifications which are usually written in a metalanguage designed to define the syntax of the language, while potential users often base a language evaluation on the benchmark approach, which involves programming representative test programs in the candidate languages and measuring such quantities as programming time, memory requirements, and execution speeds.

In contrast to the formal linguistic approach to language comparison and the experimental benchmark approach, a generalized structure of basic language constructs has been developed, within which each of the subject languages is described. In this manner, the language features are delineated within a consistent and cohesive framework and may be compared on an 'item-by-item' basis.

Since no one person within The MITRE Corporation was found to be conversant in all of the languages, a programmer/analyst, experienced in theory and practice, was chosen to provide the descriptive technical information for each of the languages under consideration.

No attempt has been made to assess the languages in a quantitative way and it remains for each potential user to either formally or intuitatively derive the criteria for evaluation, which must be

based not only on the language capabilities and the domain of the applications, but on such considerations as compiler availability and efficiency, programmer training, and the software support provided by the vendor.

SECTION III

LANGUAGE EVALUATION STRUCTURE

As previously stated, the methodology for the language evaluation consisted of the development of a Language Evaluation Structure with two major components, namely, a Language Feature Outline and a Language Evaluation Questionnaire. The Language Feature Outline is presented as Exhibit 1 and the detailed technical descriptions of the subject languages relative to the outline are presented in Appendix I. The Language Evaluation Questionnaire is presented as Exhibit 2 and the responses to the questionnaire are presented in Section V.

Exhibit 1

Language Feature Outline

A. BASIC ELEMENTS

1. CHARACTER SET
2. IDENTIFIERS
   - Variables
   - Labels
3. LITERALS/CONSTANTS
4. OPERATORS
   - Computational
   - Relational
   - Logical
   - Data Movement
   - String
   - Editing
5. KEY WORDS/RESERVED WORDS
6. PUNCTUATION

B. DATA TYPES AND ORGANIZATION

1. DATA TYPES

   a. Problem Data

      - Numeric
      - Logical (Boolean)
      - String
      - Status

   b. Program Control Data

      - Label Data/Switches
      - Procedures
      - Pointers/Offsets
      - Multitasking Data

2. LOGICAL DATA ORGANIZATION

   - Arrays
   - Structures/Tables
   - List Structures

3. PHYSICAL DATA ORGANIZATION

   - Packing
   - Alignment
   - Overlays
   - Records

Exhibit 1 (concluded)

C. PROGRAM STRUCTURE

1. UNITS

a. Non-executable Units

(1) Comments/Remarks
(2) Declaration/Specification Statements
  data declarations
  file declarations
  format descriptions
  procedures, subroutine, and function definitions
(3) Storage Allocation/Segmentation
(4) Environment/Operating System Descriptions

b. Executable Units

(1) Expressions
  subscripts/arguments
  formation rules
(2) Statements
  (a) data manipulation
  (b) program sequence control
  (c) input/output
    stream-oriented transmission
    record-oriented transmission
  (d) debugging statements
  (e) operating system interface
    multitasking/asynchronous processing
    dynamic storage allocation
    interrupts/error control
    interactive processing
(3) Compound statements
(4) Loops
(5) Blocks/Paragraphs
(6) Functions
(7) Procedures/Subroutines

c. Compile-Time Features

source text manipulation
language rule modification
initialization
inclusion of other languages

2. ORGANIZATION

a. Program Format

b. Scope of Names

Exhibit 2

Language Evaluation Questionnaire

A. Applicability

Discuss the suitability of the language to the following application areas with reference to specific language strengths and weaknesses:

a. scientific
b. data management
c. business applications
d. artificial intelligence
e. real-time systems
f. systems programming

B. Descriptiveness

Discuss the form of the subject language in terms of:

a. conciseness vs. naturalness
b. ease of writing vs. self-documentation

C. Ease of Use

Discuss the language in terms of ease of learning and use:

a. professional vs. non-professional programmer training
b. natural language subsets (modularity)
c. consistency and strictness of rules
d. ease of coding, debugging, maintaining, and updating programs

D. Transferability/Machine Independence

Discuss the language in terms of:

a. transferability
b. access to machine-specific features

E. Implementation

Discussion the available compilers for the language in terms of:

a. computer systems
b. language of compiler
c. efficiency
d. diagnostic aids
e. optimizing techniques

F. Managerial Aspects

Discuss the subject language in terms of:

a. maintenance/level of support
b. standardization/extendability
c. available documentation

SECTION IV

AN OVERVIEW OF THE LANGUAGES

AED (Automated Engineering Design)

AED, developed at MIT under the direction of D. T. Ross, with industry participation and Air Force sponsorship, was designed to support a software engineering discipline; that is, the construction of modular, reusable, machine-independent software components which can be integrated to fabricate software systems over a wide range of application areas. As such, it has evolved as a 'system of systems for making systems'. However, in its present state of development, the primary orientation is toward systems programming; i.e., the development of compilers, interpreters, and operating systems.

The AED system consists of three basic components:

(1) The AED-0 language, a derivative of ALGOL-60, is a high-level, general-purpose programming language based on Ross' Algorithmic Theory of Language.

(2) The AED library consists of integrated packages of routines which perform generalized functions relative to system programming, such as memory management, data structuring, and device-independent I/O control. In keeping with the philosophy of software engineering, a package is constituted of various subatomic, atomic, and molecular functions which share common global variables and data structures and may be integrated to accomplish logically complex operations.

(3) Finally, AED consists of two systems to automate the construction of language processors, each of which has an older (public AED) and a newer (SofTech-proprietary) form. The Finite State Machine (FSM), formerly Read-A-Word (RWORD), accepts as input descriptions of the items or words allowable in a language and generates a lexical processor (an AED-0 program) which reads a string of characters and combines them into parsing items according to the user-defined regular expressions. The Syntax Definition Facility (SDF), formerly AED Junior (AEDJR), accepts a BNF-like grammar specification and constructs tables for parsing input strings.

The system-building aspect of AED is well beyond the scope of this paper, but was included in the interest of completeness. In the context of the language comparison, the discussion of AED will be limited to the AED-0 language, with reference to the packages only to the extent that they provide the semantics of language features not included in basic AED-0.

A concept central to AED is that of the 'modeling plex', which provides the mechanism within which the relationships between the data, structure, and algorithm which define a process may be expressed. The data structure of a plex, which is defined as an interconnected set of n-component elements, is a list structure of arbitrary complexity in which pointers to other data structures are stored. The AED referencing scheme allows the components, data, structure, and algorithm to be used interchangeably at various stages in the program; for example, an algrorithm may appear as data at some higher level and may be structurally tied to some data source, which in turn may, at a lower level, be either an ordinary datum or a plex itself.

The Algorithmic Theory of Language includes the principles of type transformation and phrase substitution. Specifically, operations on data types result in the formation of larger units, with an associated data type, which can be substituted wherever a more primitive form of the data type is allowed.

Part of the AED syntax is based upon a concept called 'universal reference notation'. In this notation, all qualified references are of the form:

component (element)

where component and element may be mapped into the following implementations:

component	element
function name	function argument list
bead component	pointer
array name	subscript
macro name	macro argument list

Thus, the formal reference may be coded before the mechanization is known, or the mechanization may be changed by merely altering the declaration.

COBOL (COmmon Business Oriented Language)

The Conference on Data Systems Language (CODASYL) was established in May 1959 at a Pentagon meeting consisting of representatives of user groups, government installations, industry, and computer manufacturers, and was charged with the responsibility to investigate the feasibility of developing a common business-oriented language. As a result

of this activity, the initial specifications for COBOL-60 were developed. The language was designed to have a narrative, English-like syntax, to be machine-independent, and to provide constructs and data structures particular to business and commercial data processing, which, in contrast to scientific data processing, is characterized by the manipulation of large files of similar data records, requiring relatively simple computational and logical operations, and by the generation of rigidly-formatted reports.

It was the intent of CODASYL that the specification and maintenance of COBOL be a continuing effort in order to ensure that the language be refined as necessary for the resolution of inconsistencies, and updated to reflect advances in computer hardware and software technology.

The first revision to COBOL, known as COBOL-61, involves organizational changes to the language rather than the addition of any major functions.

COBOL-61 EXTENDED introduced the SORT feature and the REPORT WRITER option; COBOL, EDITION 1965, the most widely implemented version of the language, provided facilities for the processing of mass storage files and a TABLE HANDLING feature with indexing and search options.

COBOL, 1968, is especially significant because of the inclusion of inter-program communication (via the CALL verb and the LINKAGE Section of the DATA Division) and the concept of a run-unit, consisting of previously-compiled program modules linked together at object time.

The description of COBOL in this report is based on COBOL, 1968; however, recent COBOL enchancements include the addition of a Communications Facility, whereby a COBOL program can communicate with message-handling devices, a Debugging Facility which allows for the enabling or disabling of 'debug code' at both compile time and object time, and a Merge Facility which provides a corollary for the Sort Facility provided in earlier versions.

The responsibility for future language extensions resides with the Programming Language Committee (PLC) of CODASYL and will be reported via the Journal of Development. The primary orientation of future language extensions is reflected by the existence of the Data Base Task Group, the Asynchronous Task Group, and the Mass Storage Task Group within the PLC.

The definition of the current United States COBOL standard was accomplished in 1968 by the United States of America Standard Institute (USASI). Named the American National Standard COBOL (ANS or ANSI COBOL) and published as USA Standard X3.23-1968, it is based primarily on COBOL, EDITION 1965, plus language enchancements approved through 1966. The Journal of Development 1970 contains the current state-of-the-art COBOL specifications, a modularized subset of which will form the basis for a revision of the current standard.

ANSI COBOL is organized into a set of eight Functional Processing Modules (FPM); each module is divided into two or three levels of decreasing power. These modules and the associated levels are presented in Table I.

Table I

COBOL Functional Processing Modules

MODULE	LEVELS
1. Nucleus	HIGH, LOW
2. Table Handling	HIGH, MID, LOW
3. Sequential Access	HIGH, LOW
4. Random Access	HIGH, LOW, NULL
5. Sort	HIGH, LOW, NULL
6. Report Writer	HIGH, LOW, NULL
7. Segmentation	HIGH, LOW, NULL
8. Library	HIGH, LOW, NULL

Currently, Minimum Standard COBOL is defined as consisting of the lowest level of each module. Excluding the null cases, the standard becomes the minimum levels of Nucleus, Table Handling, and Sequential Access. A full ANSI COBOL must contain the maximum (HIGH) level of implementation of all eight FPMs.

This categorization by function and levels within functions provides a consistent mechanism whereby specific implementations may be accomplished.

In accordance with the objective to maximize compatability across many machine configurations, COBOL specifies a program structure which isolates computer-dependent information, data descriptions, and the processing procedures. In particular, a complete program has four major units called DIVISIONS which may be further partitioned into SECTIONS.

The major functions of the divisions are outlined briefly:

IDENTIFICATION DIVISION -- identifies the source program and the resultant compilation output. The program name is required;

optional entries may include programmer name, date, security level, etc.

ENVIRONMENT DIVISION -- contains all the machine-specific information relative to the configuration of the compiling and object computers; provides for the cross-referencing of hardware units with program mnemonics; establishes the correspondence between data files and the external media; specifies file control information and the input-output techniques.

DATA DIVISION -- identifies and describes each item of data to be processed. The DATA DIVISION consists of the following SECTIONS: FILE, WORKING-STORAGE, CONSTANT, LINKAGE, and REPORT. The FILE SECTION describes the content and organization (physical and logical) of all files; the CONSTANT and WORKING-STORAGE SECTIONS provide for the specification of user-defined constants and internally-generated data; the LINKAGE SECTION describes external data, references to which must be resolved at object time; and the REPORT SECTION specifies the content and format of the output reports.

PROCEDURE DIVISION -- contains all the executable operations necessary to accomplish the problem solution. The division consists of declaratives and procedures. The English-like nature of COBOL is most obvious in this division where verbs specify actions and are combined with operands to form statements, which are categorized as imperative, conditional, or compiler-directing. Higher-level executable units consist of sentences, paragraphs, and sections.

In summary, the divisions ranked in order of increasing machine dependency are: IDENTIFICATION, PROCEDURE, DATA, and ENVIRONMENT.

JOVIAL (Jules Own Version of the International Algebraic Language)

JOVIAL, a general-purpose, procedure-oriented programming language derived from ALGOL-58, was designed and implemented by the System Development Corporation in 1959-1960 for the programming of the Strategic Air Command Control System. It has since become an SDC corporate language standard, the Air Force standard for Command and Control Systems, and the Navy standard for Strategic Command Systems.

JOVIAL combines numerical scientific facilities with capabilities for data handling and the manipulation of logical entities, and, most importantly, supports the COMPOOL (COMmunications POOl) concept, whereby descriptions of system elements (data and storage allocation parameters for the data and for the system programs as well as procedure definitions) may be standardized and centralized for common reference. The COMPOOL concept provides the mechanism to separate the system-wide set of declarations describing the system data to be processed from the set of statements defining the processing algorithms. The magnitude of large-scale, computer-based information processing systems predicates such a common source of data description in the interest of both data design coordination and program transferability.

To provide for the efficient transfer of programs and to minimize conversion costs, JOVIAL compilers have a two-phase structure consisting of a Generator and a Translator, each of which is written in JOVIAL and performs a distinct transformation. In particular, the Generator accepts as input JOVIAL declarations, source statements, machine-language code, and the COMPOOL declarations; codifies the data description declarations and determines appropriate sequences of

elementary operations; and, finally, produces as output a machine-independent Intermediate Language (IL) consisting of a dictionary and an operation list. It is the function of the second-pass Translator to convert the Intermediate Language to machine-specific code; translators ordinarily incorporate a complete symbolic assembly phase.

The efficiency of this method derives from the fact that all compilers use the same Generator and the writing of a new compiler reduces to the less complicated task of writing the Translator. Also, a common Generator provides for the centralized control of the grammar of the language.

JOVIAL has gone through a series of revisions and currently exists in several dialetical forms. The original JOVIAL-1 was replaced by JOVIAL-2 which was in turn superceded by JOVIAL-3 (J-3), the latest official version of the language. J-3 forms the basis for the current Air Force standard as specified in Air Force Manual 100-24. However, this standard is being revised and the specification for a new standard is due for publication in the fall of 1972.

Current SDC policy specifies that JOVIAL compilers must implement a 'core subset' of J-3, called Basic JOVIAL or JS; all implementations of J-3 specifications must be in accordance with the corporate standard; and the addition of advanced language features must be compatible with the standard.

J-4, J-5, J-5.2 and J-5.3 represent supersets of J-3, while JTS is a time-sharing version of Basic JOVIAL.

PL/I

In October 1963, the Advanced Language Development Committee of the SHARE FORTRAN Project was formed under the sponsorship of SHARE and IBM for the purpose of defining a new high-level programming language applicable to a wide spectrum of applications, i.e., scientific, commercial, real-time, and systems programming. The design criteria further specified that the language should provide full access to machine and operating system facilities while maintaining relative machine independence; it should be modular in nature allowing natural subsets of the language for particular applications and differing levels of programmer expertise; and, finally, it should be 'forgiving' by providing a default interpretation for each of the many options.

Originally envisioned as a compatible extension to FORTRAN, the language, as it evolved, synthesized features from FORTRAN, COBOL, and ALGOL and, additionally, incorporated new constructs in support of the architecture of third-generation computer systems.

Reflecting this advance in the state-of-the-art, the original version of the language was designated NPL, The New Programming Language. However, this acronym was abandoned in 1965 in favor of 'PL/I' which, although the official name of the language, is derived from Programming Language/One.

Historically, the first language specification was defined in March 1964 and the first compiler for System/360 was released in August 1966. Subjected to major revisions, additions, and deletions through its development, the language is presently in the process of formal standardization. The American National Standards Institute (ANSI X3J1), in cooperation with the European Computer Manufacturers

Association (ECMA TC10) and with the technical cognizance of the International Federation of Information Processing (IFIP Technical Committee 2), is working toward the definition of an international standard for PL/I to be completed by December 1972.

PL/I is a language of great power and flexibility; its more advanced language features include provisions for:

1. interrupt handling,
2. asynchronous processing,
3. dynamic control of storage allocation,
4. list processing, and
5. compile-time macro facility.

Error control and program checkout are greatly facilitated by the interrupt-handling features of the language which allow for the specification of programmer-defined corrective action to be taken when dynamic, unscheduled conditions such as overflow, end-of-file, subscript range, and conversion errors occur. The asynchronous features provide for program operation in a multi-programming environment; routines may execute in parallel and operations may be suspended to await the occurrence of a specified event. Storage management is explicitly provided for by the use of attributes which define the manner in which variables will be assigned memory space; options include permanent, block-dependent, and programmer-controlled. The utility of PL/I as a list processing language is enhanced by the provision for complex data structures, the definition of pointers, offsets and based variables, and facilities for the dynamic control of storage.

The preprocessor phase of a PL/I compilation provides the mechanism to manipulate a source program prior to the actual generation of

object code; included are facilities for macro definition, common text insertion, and a limited syntax (e.g., assignment, GOTO, IF, DO).

Of current interest in the development of PL/I is the announcement by IBM of both a Checkout and an Optimizing Compiler. The Checkout Compiler, useful during program development, generates interpretive code and provides diagnostic error messages in source language terms. In addition to checking for such deviations as the use of uninitialized variables, illegal pointer references, and illegal branches, the Checkout Compiler includes methods for program monitoring, a spelling correction algorithm for key words, and a formatting program which generates source program listings in a structured schematic form. The Optimizing Compiler, designed specifically for the generation of efficient object code, performs program flow analysis to accomplish such ends as the removal of constant expressions from within interactive code, the consolidation of common subexpressions, and the local optimization of register allocation and use. The compatability of the Checkout and Optimizing Compilers in both source language and object code supports the concept of efficient modular program development.

SECTION V

RESPONSES TO LANGUAGE EVALUATION QUESTIONNAIRE

APPLICABILITY

Language capabilities must be evaluated in relation to specific applications areas. No one language can be considered best for all applications. In fact, this is contrary to the design objectives of most current procedure-oriented languages, which, although designed to provide a wide range of programming capabilities, are usually oriented to a specific application area. Clever programming techniques can often compensate for language deficiencies, but they usually do so at the expense of efficiency.

AED

Since AED provides easy access to machine-specific features, it is quite adequate for its declared purpose, viz. 'system programming' in the sense of constructing operating systems, utilities and such. It has been applied successfully to compilers, despite relatively poor string-handling facilities. Heuristic programs, including 'artificial intelligence' and other applications requiring complex data structures, would also seem to be appropriate.

Lack of matrices, complex or double-precision data types would seem to limit scientific applications, while poor string-handling and I/O would tend to discourage data management or business applications. Real-time systems would require machine language support for interrupt processing.

COBOL

As stated in the history and as is obvious from the name of the language, COBOL is a business-oriented programming language. The basic language features of COBOL, in the DATA Division, provide a natural way to manipulate, update, change, and add data to a variety of record formats and file descriptions. Built-in Sort, and Table Handling facilities, asynchronous processing, and report writing are effective tools of the language for data management and business applications.

As the language has been modified, new features have allowed COBOL to be used in other applications areas with a reasonable amount of success.

COBOL is not recommended for applications that are heavily scientific involving such operations as matrix manipulation, inversions, iterations, and transformations. These applications may be programmed; however, much of the computation would best be performed by a call to a more appropriate language via the ENTER or CALL verbs. COBOL, however, is suitable for simple statistical and probability problems.

The language itself is not prohibitive to real-time systems and systems programming applications, but the excessive number of requirements outside the scope of the language would make it impractical.

JOVIAL

JOVIAL is probably best suited to Data Management problems and, in particular, Command and Control applications, where JOVIAL's emphasis on logical processing (especially status data) would be useful.

The COMPOOL was specifically designed for multi-programmer implementations, which is characteristic of most Data Management problems. Although data can be packed for efficiency, JOVIAL is weak on data structuring; hierarchical structures and dynamically based data cannot be handled explicitly in the language.

JOVIAL is only moderately suited to scientific problems. Arithmetic operations are limited to individual items (in contrast with PL/I, for example, with matrix operations) and the built-in functions are directed almost entirely either at bit manipulations or at table size definitions -- the only arithmetic function is ABS. Multidimensioned data is only available when the ARRAY statement is implemented and double-precision values are unknown.

The biggest deficiency of JOVIAL for business applications is the lack of a formatted I/O capability for report generation; repeating groups are available with the STRING primitive. JOVIAL's arithmetic processing capabilities would be entirely adequate in most cases, and its logical processing capabilities would be useful also. The COMPOOL would be useful in centralizing and standardizing file and item declarations for a full system.

Since JOVIAL does not provide for tree structures for data, dynamic control of data allocation, recursion and stacks for programs, and automatic garbage collection, it would be inadequate for Artificial Intelligence problems.

While JOVIAL applications frequently operate in a real-time environment, time is involved only as another piece of data. JOVIAL as a language has no provision for processing interrupts or otherwise responding in an asynchronous fashion -- this is normally done

by the supporting monitor. In other words, JOVIAL is designed for programs which operate in a real-time environment, but not for programs that deal with a real-time environment. The difficulties in going from one type of operation to another could be dealt with by breaking into DIRECT code (working at the machine-language level), but JOVIAL's DIRECT code capability seems more a way of overcoming mismatches between language and application than a part of the language itself.

Systems Programming involves dynamic management of hardware resources, and JOVIAL tends to stay above hardware-specific problems (except for data packing). Queues, stacks, and dynamic arrangement of storage are not supported by the JOVIAL language, and the objections to JOVIAL for Real-Time Systems apply as well as to its use for monitors and operating systems. However, JOVIAL has been used successfully to write its own compiler.

PL/I

PL/I, unlike most languages which were created for a specific application, was designed to provide generality and flexibility in order to handle many applications well. The ability to chain variables and structures together with pointers and to dynamically allocate and free storage, allows PL/I to act as a good list processing language. The use of PL/I for systems programming is enhanced by the modularity of object code, ON-conditions, recursive procedures, dynamic storage allocation, varying length strings, and the ability to access data at the bit level of the machine. Complex and double-precision operations, plus matrix-handling capabilities, and a large library of routines make PL/I useful for scientific applications. The ON-conditions provide for some interrupt handling necessary for

real-time applications. Slow I/O limits the capabilities of PL/I for business use, although the speed of I/O has been doubled by IBM's Optimizing Compiler. Some language deficiencies include graphics, formula manipulation, and complicated pattern matching.

DESCRIPTIVENESS

The form of programming languages can vary widely. At one end of the spectrum are the very cryptic symbolic assembly languages and at the other, the very natural English-like languages. Conciseness is an advantage in that code may be generated with a minimum of physical writing, but the operations being performed are often obscured and difficult to follow. The English-like languages are apt to be cumbersome to write but have the advantage of producing programs which are self-documenting and therefore easier to maintain and update.

AED

From the point of view of pure syntax, AED strikes a reasonable and conventional balance between conciseness and naturalness and may thus claim both. The underlying syntax is simple and regular and therefore concise; standard infix arithmetic and Boolean expressions are used rather than 'English'. On the other hand, English function names rather than new operators tend to be used for non-elementary operations. Unfortunately, when principle is reduced to practice, the use of the language and its standard library requires much obvious artificiality to overcome semantic shortcomings. The sequence of coding PACK statements, performing the required setup for a subroutine package, and referencing acronym-named procedures in these packages, produces results that are usually neither concise nor

natural. The main difficulty in reading or writing AED is in knowing the nature and order of parameters to be passed to standard package routines. AED is definitely not self-documenting.

COBOL

COBOL was designed to follow closely the syntax of the English language. By virtue of this design specification, the language evolved as more natural than concise. Because of the restrictions on format and reserved words, COBOL presents difficulties in writing. It is a very wordy language, yet this very wordiness (somewhat diminished by the ability to abbreviate certain reserved words) does much to achieve a reasonable level of self-documentation. Perhaps COBOL's greatest assest is its structured organization -- enabling distinction for documentation of program identification, environment, data, and procedures.

JOVIAL

JOVIAL is a concise rather than a natural language. All statements except assignment statements start with reserved key words. and statements are generally evaluated independently of neighboring statements; (IF, IFEITH, ORIF are exceptions). The introduction of compound statements (BEGIN...END) and free format source language statements are important contributions to naturalness, as is the use of name-type labels. On the other hand, the limitation of identifiers (8 characters in some implementations) contributes to conciseness rather than naturalness. The major language feature which supports both conciseness and naturalness is the DEFINE statement. Conciseness is helped by the use of a single identifier to represent any arbitrary string of text; for example, PRINT could be used in

place of PRNT (LINE, 15). Naturalness is helped by replacing expressions with mnemonically useful identifiers; for example, SCANIN could be used to represent FOR I=Ø, 1, ALL(IN) $ BEGIN in processing an item called IN. Finally, naturalness is helped considerably by the use of status values (e.g., V(OFF)), where text is used to represent number values.

JOVIAL is an easy language to learn, only a little more difficult than FORTRAN (primarily because of more complex data structuring). Once learned, it is relatively easy to code, in part, because many of the data declarations can be put in a COMPOOL and, in part, because of features like the CLOSEd procedure which are consistent with structured programming techniques.

JOVIAL is a relatively poor self-documenting language, comparable to FORTRAN in this regard, except that comments may appear wherever blanks are legal.

PL/I

PL/I has tried to achieve a balance between conciseness and naturalness but appears to lean toward conciseness. With no knowledge of programming one would find it difficult to understand a PL/I program, but with only a basic knowledge of almost any higher-level programming language most PL/I statements could be understood. Varying degrees of conciseness or naturalness may be attained through the use of macro statements; for example, arithmetic operators may be replaced by their English equivalant -- PLUS→ +. The language is relatively easy to learn, only slightly more difficult than FORTRAN. Although PL/I is not a self-documenting language, documentation is enhanced by the ability to insert comments anywhere that a blank is allowed.

EASE OF USE

Of further consideration in the evaluation of programming languages is the ease of learning and use. A distinction must be made between the professional and the non-professional programmer. The former is usually associated with large-scale systems and is interested in sophisticated programming techniques and the generation of 'elegant and efficient' code; while the latter is more apt to want answers to 'one-shot' problems and to be satisfied with a 'quick and dirty' solution. The general utility of a programming language is enhanced if it can be modularized to provide working subsets of the language to satisfy the requirements of a wide spectrum of users. Indeed, the novice programmer does not even have to be aware of the full power of the language.

AED

A one-week training course, provided by SofTech, Inc., is considered sufficient for a general introduction to the AED language. A working knowledge of the language and some of the more important tools would require a two-week training course. Additional training and/or consultation with SofTech would be indicated for a large system application.

AED does not lend itself to subsetting at the language level, being somewhat skeletal as it is. With regard to the standard packages, use of any package is optional and individual applications typically use some relevant subset of these.

As noted previously, AED systax is consistent at the most basic level, less so as one approaches the surface, and often quite inconsistent when the interface requirements of the various packages are considered.

Aside from the language deficiencies previously discussed, AED presents no particular coding difficulties. Debugging may be undertaken in conventional ways, including dumps, with no particular difficulty. A TRACE package permits elaborate instrumentation at the point of procedure CALL/RETURN. An important feature of this package is that it may be applied to a running system without recompilation; it simply replaces a standard procedure linkage routine.

Maintenance and updating (functional enhancement or tuning) of programs is generally facilitated to the extent that the AED philosophy of separating algorithm, data, and structure is possible and implemented. The universal reference notation is one aspect of this.

COBOL

The original concept of COBOL was to make it easy to learn by non-professionally trained programmers. As the language has evolved, the need for professional programmer training has increased in order to adequately provide for the use of such features as the high-level interactive capabilities, random access input/output, and the new data manipulation facilities.

COBOL is very strict and very consistent in its rules. The requirement of data definition and description prior to use in the PROCEDURE Division results in the generation of consistent code. As a standardized language, the rules must be strict or the standardization

has no meaning. Most rule violations (e.g., format, specifications) are detected at compile time.

Coding in COBOL is not necessarily easy; it is time-consuming because of its verboseness. Coding does become easier once the reserved words and format restrictions are completely understood. COBOL does not provide the facility for direct debugging statements. Structuring violations and data incompatibilities detected at compile time frequently serve to reduce the amount of debugging necessary at execution time. Major logic problems usually require coding of intermediate output or flags to determine a trace of the program during execution.

COBOL programs are relatively easy to maintain and update -- the rules, format, organization, and labeling process facilitate this aspect of programming.

JOVIAL

Once familiar with basic programming concepts, which are represented in varying ways in most higher-level languages, JOVIAL programmers are easy to train; this in part accounts for the extensive use of JOVIAL both in the Air Force and in the Navy.

Many subsets of JOVIAL have been and are still being developed. J-5.2 is the superset which nobody implements; J-3 is the AFM 100-24 subset; J-S is SDC's own 'core' subset. Each compiler seems to be implemented with minor language differences. File I/O (which is the only I/O JOVIAL recognizes), for example, seems to be one of the first features dropped.

There are some inconsistencies regarding what constitutes a 'statement'. For example, the BEGIN and END symbols are thought of as delimeters in their own right and so do not require the $ terminator. Another inconsistency is the GOTO, which may go to a SWITCH, may go unconditionally to a label and not return, or may go to a CLOSEd procedure and return when the procedure finishes. The major inconsistency therefore is that a GOTO does not always effect an absolute transfer of control; all cases, however, are coded as:

GOTO < ident > $

and cannot be distinguished in local context except by comment.

The most useful debugging tool is the set/used listing provided with support software. The COMPOOL and the associated support software are important maintenance features. JOVIAL's disregard for I/O, especially formatted I/O, will increase debugging problems. Data dumps and tracing, for example, would have to be coded in-line or provided by test drivers.

PL/I

The basic PL/I capabilities (i.e., those comparable to FORTRAN) can be learned by most programmers in a fairly short period of time, but because of the complexity of PL/I, the real power of the language (structures, pointers, based and controlled variables, etc.) will take a longer time to master.

PL/I is not a very consistent language in the sense that many exceptions exist. Many statements, especially I/O statements, have strict rules governing their content, but declaration statements, format statements, and internal procedures may be placed anywhere in

a block. Almost all types of data conversions are made by the compiler when assigning one data type to another.

Debugging tests include the CHECK and SUBSCRIPTRANGE prefixes which print the value of a variable each time that value changes and checks for subscripts outside the allowable ranges in arrays. Coding and updating are relatively simple because of the balance between conciseness and naturalness.

TRANSFERABILITY/MACHINE-INDEPENDENCE

Machine-independence is a major goal of most high-level programming languages. To the extent that a language is free of machine-specific constructs, it is considered to be transferable; i.e., programs written in the language may be executed on any computer system for which a compiler exists. Since the cost of programming has been the dominant factor in the utilization of computer systems, portability of computer programs becomes a prime economic concern.

AED

Explicit machine dependencies in AED are generally due to the fact that the concept of a machine 'word', although not a data type, nonetheless permeates the language. This is evidenced in type declarations (for example, the range of an INTEGER will depend on word size), allocation of bead components to words or packed portions of words, operations on groups of bits, and the I/O routines. Implicit machine dependencies occur when the program relies on overlaying -- setting pointers or otherwise arranging to have a single datum in the machine treated as two or more items of different type in the program. One method of doing this -- a method actually recommended and referred

to as 'legal pornography' -- is to take advantage of the lack of type checking when arguments are passed and to pass arguments unlike the corresponding parameter in type.

Except for packing the machine word, AED does not allow access to special machine features. A call to the assembly language would be required.

With careful use of the language, the problems of transferability can be reduced to changing declarations, macros, or function mechanizations in a few central INSERT files (see Appendix I, Section C.1.c). This is the technique used in bootstrapping the AED compiler itself.

COBOL

Source programs in COBOL, conforming to the COBOL standard specifications, are readily transferable with minor changes in the ENVIRONMENT Division. Caution must be exercised, however, since COBOL has several versions and some elective features may be unavailable on different machine implementations. If the standard COBOL is assumed, then the minimum requirements will be available on all machines supporting standard ANS COBOL. Manufacturers usually stipulate what version of COBOL is supported and at what level -- FULL or LOW ANS COBOL.

COBOL presents standard specifications and allows the implementator to augment or eliminate certain specifications based on the capabilities of the machine. Through the ENVIRONMENT Division, all necessary input/output control can be made machine specific. Other machine-specific features may be incorporated.

JOVIAL

The transferability of JOVIAL programs was enhanced with the

publication of AFM 100-24 and JOVIAL's selection as the standard AF Command and Control Language. In programs in which data are structured or packed, transfer to computers with different word sizes or memory organizations presents problems. These can be anticipated and solved in part, particularly for large program systems, by judicious use of COMPOOLS and DEFINE statements. In general, JOVIAL is a reasonably transferable language with the obvious exception of DIRECT code. JOVIAL's limited I/O capability at the source level eliminates some problems associated with transferability; however, machine-specific I/O subroutines must be provided.

JOVIAL has good access to certain machine-specific features. Here too, the most obvious feature is DIRECT code which permits assembler-level coding of programs (a practice to be avoided in most cases). JOVIAL's data-packing capability is another feature emphasizing machine specificity. On the other hand, JOVIAL's general lack of language-level I/O is extremely non-specific.

PL/I

While PL/I is fairly machine-independent, it does have some machine-dependent features. The UNSPEC command which gives the bit configuration of a variable is obviously machine -- as well as possibly compiler -- dependent. Overlaying of variables, whether accomplished by the user with pointers, or by the compiler with the DEFINED statement, may be machine-dependent, contingent upon the internal data organization. The size of a variable is also machine-dependent (e.g., the declaration BIN FIXED (17) on the IBM 370 would actually result in the allocation BIN FIXED (31)). Implementations to date guarantee only that the number of bits requested will be treated as the minimum for allocation.

IMPLEMENTATION

Given that languages are not completely standardized and exist in many versions and dialects, the problem of language evaluation is further complicated by the many variations due to compiler dependencies.

The evaluation of compilers is beyond the scope of this paper and is indeed a subject unto itself. This section is included, however, to place the subject languages in context and to give some general indications regarding the availability of compilers and their relative efficiency.

AED

AED compilers exist for the IBM 7094, the UNIVAC 1108, the CDC 6000 series, and the IBM 360/370 series (360/40 or larger) under OS, DOS, TSO, and the CP/67 derivative time-sharing virtual machine systems. An interactive version is offered by National CSS. Cross-compilers exist from the 360 to the IBM 1130, Honeywell series 16, Raytheon series 700, and the DEC PDP-10. The various implementations of AED exhibit an unusual degree of compatibility, because of the transfer method used (bootstrapping) and also because the same implementation team (SofTech) generally performs the transfer.

The AED compiler is said to be 93% coded in AED, with 83% of the system machine-independent. A cross-compiler costs about $30K, and a full one about $150K.

The 360 AED compiler seems to be roughly comparable to the PL/I (F) compiler in speed of object code produced, though much worse in space requirements because of the heavy reliance on libraries. Of course, efficiency will vary with programs and programmers: SofTech's figures show PL/I (F) uniformly worse in space and varying from somewhat better to four times worse in speed.

The literature does not discuss optimizing techniques within the compiler. However, they appear to be conventional and do not provide for such features as code rearrangement.

COBOL

COBOL is at this time available on almost all second and third-generation equipment. The versions of COBOL, however, range from COBOL-61 to COBOL-68 EXTENDED. Manufacturers have implemented versions of COBOL based on need and availability of core storage. The core requirements for a COBOL compiler range from 8K to 32K.

The language used to write the compiler as well as the compiler efficiency are factors associated with each specific implementation. Several implementors have special COBOL translators as well as compilers.

JOVIAL

Computer systems for which a version of JOVIAL is available include IBM 360/50, UNIVAC 1108, CDC 3600, CDC 3800, CDC 6600, H4400, HIS 6000 (WWMCCS), PDP-10, and various militarized computer systems.

With a number of exceptions, JOVIAL compilers are written in JOVIAL so as to be self-compileable. Of course, significant portions (limited by AFM 100-24 to 5% of the object program) may be in DIRECT (machine language) code.

Efficiency requirements are stated separately from AFM 100-24 for each compiler procurement.

Diagnostic aids include source program listing, library program listing, object program listing, and environment listing (these four are mandatory), and set/used item listing, runtime program error monitoring, alter mode, alter-update mode, and grammar-checking mode (these five are optional).

Optimizing techniques are not standardized and will, if supplied, vary with individual implementations.

PL/I

PL/I compilers exist on IBM 360/70 series machines, Multics GE645, and Burroughs 5500/5600 series. CDC is currently working on an interpretive PL/I compiler for the future. The IBM compiler is written in assembly language and the Multics compiler is in PL/I. An interactive PL/I subset is also supported by IBM. IBM has recently released an optimizing PL/I compiler which reduces core allocation and increases execution speed at the expense of compile time.

MANAGERIAL ASPECTS

Beyond the purely technical characteristics of a programming language, of importance also are the associated managerial aspects. Such considerations as the maintenance of the language and the level of support, the degree of standardization or (conversely) the potential for extensions, and the availability and quality of the documentation must be evaluated before the impact of any long-term commitment to a programming language can be assessed by an installation.

AED

The maintenance, extension, and documentation of AED resides

with SofTech, Inc., in cooperation with a user group. Existing documentation, especially on the all-important packages, is poor, necessitating extensive consultation with SofTech at least for the initial implementations. No 'official' standards are set up or projected; the latest implemented compiler may be considered a de facto standard.

COBOL

As COBOL grows in use through commercial applications, the level of support and maintenance is increasing. Most implementors routinely provide compilers for their new product lines and, in particular, COBOL is gaining wide acceptance on the mini-computers.

COBOL is constantly being evaluated and improved. The standard version serves as a basis for most implementations. The overall structure of COBOL is not changing; however, proposed changes include enhancements in the area of data manipulation and the increased use of abbreviations for reserved words.

Implementors provide machine-specific documentation. Reports have been published by CODASYL defining the language from version 60 through version 68. New publications produced by the same group describe the data handling extension capabilities.

JOVIAL

The JOVIAL language, as opposed to compilers implementing the language, is chiefly supported by System Development Corporation. The Air Force maintains the J-3 version of the language with AFM 100-24, which is soon to be revised. The SDC language, J-5.2, is a

superset of characteristics which has never been implemented in its entirety, but which serves as a starting point for actual implementations.

Standardization is reasonably good in that there are documented descriptions of official versions.

As far as extendability goes, the tendency is to subset the J-5.2 version for implementation rather than to extend the language. Contractors working from AFM 100-24 generally provide compilers which depart to greater or lesser degrees from those specifications, which therefore tend to serve as a starting point.

Standardization has been a problem with JOVIAL in that each compiler tends to have minor deviations from the others. AFM 100-24 was published to combat this, but complete uniformity will probably never be achieved.

Documentation for JOVIAL is extensive, with each compiler documented separately. SDC's official document for J-3 is SDC TM-555; the Air Force official version is AFM 100-24.

PL/I

IBM, as one of the developers of PL/I, is giving the language full support. Documentation is readily available from IBM. Currently a joint effort by ANS and ECMA is underway to produce a standard PL/I. This effort is yet to be completed and it is speculated that the standard may be an extension of PL/I as it now exists on IBM's Optimizing Compiler.

APPENDIX I

TECHNICAL CHARACTERISTICS

Appendix I delineates the technical characteristics of the subject languages, organized in accordance with the Language Feature Outline which was presented as Exhibit 1 in Section III. Each topical section is headed by a numeric identifier and a title which establish the correspondence between the text and the outline. Additionally, an introduction is provided for each topic in order to define the terminology and context. This prefatory material is then followed by the language-specific discussions, as provided by the programmer/analysts, for the subject languages -- AED, COBOL, JOVIAL, and PL/I.

A.1 Basic Elements - Character Set

The hardware representation of a programming language consists of a unique character set from which selections are made to denote and reference data, and to specify the processing logic. As distinguished from the reference and publications languages, the hardware language, by definition, consists of a character set which is directly compatible with the computer system. The extent to which special characters exist in a language and may be used will, to some degree, limit the necessity for defining key words to specify basic operators.

AED

AED source text utilizes the 26 alphabetic characters, the digits 0-9, plus the following special characters:

. + - * / , $ ( ) = ' blank

Other implementation-defined characters may appear in comments, character literals, or be handled in I/O.

COBOL — The basic character set of COBOL consists of 51 characters:

A-Z, 0-9, + - . ; * = / ( ) < > " , $ and space.

Other characters, machine-acceptable, may be used in comments and character strings. The double quote is represented as a single quote on some configurations.

JOVIAL — The JOVIAL character set consists of the 26 uppercase letters (A, B,...Z), the 10 digits (0, 1,...9), the space, and the following 11 special characters:

+ - * / , . = ' $ ( )

PL/I — Either a 60 or a 48-character set may be used to write a source program. The 60-character set is composed of 29 alphabetic characters ($, #, @ and 26 letters of the English alphabet), ten digits, and 21 special characters:

+ - * / = , . ( ) ' < > ; % :¬ & | _ ? blank.

The 48-character set is composed of the letters A-Z, the digits 0-9, and the following special characters:

+ - * / = , . ( ) ' $ blank

In all but four cases, the characters of the reduced set can be combined to form the missing characters of the larger set. The four 'missing' characters are: @ # _ ?. Only when the 48-character set is being used are there any 'reserved words' in PL/I (e.g., LT represents <).

Table I presents a summarization of the special characters provided by the subject languages.

Table II

Special Characters

Language \ Character	+	-	*	/	=	$	,	.	(	)	blank	'	>	<	;	"	%	:	¬	&	|	_	?	@	#
AED	•	•	•	•	•			•	•	•															
COBOL	•	•	•	•	•		•	•	•	•	•														
JOVIAL	•	•	•	•	•			•	•	•															
PL/1 (60)	•	•	•	•	•	•	•	•	•	•	•														
(48)	•	•	•	•	•			•	•	•															

A.2 Basic Elements - Identifiers

An identifier is a programmer-constructed mnemonic consisting of a combination of alphanumeric and special characters assembled according to the language specifications and used in the source program to reference data elements (i.e., items, arrays, structures, strings, and files) and program units (i.e., statement labels and procedures names). The readability of a program may be considerably enhanced by non-restrictive formation rules.

AED — Labels and variables of all types are named by a string of alphabetics and/or (single) periods.

COBOL — Variables are defined as any data item referred to by name whose value may be changed. Variable names must conform to the rules for a COBOL word. A word may be 30 characters maximum, and may contain the characters A-Z, 0-9,

and hyphen/minus. The hyphen may not be the first or last character. A space is not an allowable character. All variables must be defined in the DATA Division and must be uniquely identifiable -- name unique or qualifiably unique; variable names must contain at least one alphabetic character. There are eight variable types: numeric, numeric edited, alphabetic, alphanumeric, alphanumeric edited, conditional, control, and index. Variables are considered to be contiguous or non-contiguous. The placement and coding in the DATA Division determines their grouping.

Labels must conform to the rules for a word and be terminated by a period. Special labels such as division headers and section names must be terminated with a space and followed by the word DIVISION or SECTION and then terminated by a period. Paragraph labels must be unique within a section. Paragraph names may be all numeric.

JOVIAL

Names and labels in JOVIAL are formed identically. Single-letter names are reserved for iteration vaiables. All others (both names and labels) are formed from 2 or more (implementation-limited) characters. The initial character must be alphabetic, the remaining characters may be numeric, alphabetic, or prime ('), except that consecutive or final primes are illegal. No name or label can match any reserved word (see A.5). Default attributes are assigned to names depending on the first character; different defaults can be assigned with MODE.

PL/I

Both variables and labels consist of a single alphabetic character or a string of alphanumeric and break characters, and are preceded and followed by a blank or some

other delimiter. The initial character must be alphabetic. On the 360/370 computer, the length of the string must not exceed 31 characters and when the identifier is 'external', the compiler truncates it to 7 characters by concatenating the first 4 and last 3 characters.

A.3 Basic Elements - Literals/Constants

Constants, as differentiated from symbolic data, are program data that have values which are not subject to change. A literal, a special type of constant, is a string of characters which represents itself, thereby allowing the use of its value as its name.

AED

Constants which may be represented in AED are:

1. numeric constants
   - integer
   - real
   - octal
2. Boolean constants

In general, literals/constants are defined for any data type except pointers and may appear wherever a non-storing reference to a variable of that type could appear. Integer constants have the form:

$$\text{<digits>} \left[\left\{ \begin{matrix} C \\ D \\ E \end{matrix} \right\} \text{ <decimal no.>} \right]$$

where the digits represent the characteristic in octal (C) or decimal (nil, D or E) and the decimal number is the exponent of a scale factor whose base is 2 (C or D) or 10 (E). Real constants have the form:

<digits with decimal pt.> [E [<sign>] <decimal no.>]

with a similar meaning. Boolean constants are the reserved words TRUE and FALSE.

Although there is no character data type at present, character literals may be expressed in the alternate forms:

.C. q string q
.BCD. q string q
'single - letter'

where 'q' represents an arbitrary quoting character not in the string. The type of such a constant is either POINTER (.C.) -- the address of an area containing the string, or INTEGER (.BCD. and 'letter') -- a FORTRAN-like string surrogate. Minor variants of .C. and .BCD., having to do with allowed codes and manner of storage, are also defined. Similarly, the addresses of procedures or labels, referenced LOC (procname) or LOC (label), are in effect constants of type POINTER.

COBOL

COBOL defines the following types of constants:

1. literal
   numeric
    integer
    fixed-point
    floating-point (optional)
   non-numeric
2. figurative

Some versions of COBOL, still utilizing the CONSTANT Section of the DATA Division, have named constants. These constants are named as variables but may not be changed in the program.

Constants are divided into two types, literal and figurative. Literals may be numeric or non-numeric. A numeric constant may be fixed-point, integer, or floating-point: a fixed-point constant is composed of 1-18 digits, an optional sign, and a decimal point in any position except the rightmost; an integer constant is composed of 1-18 digits, an optional sign, and no decimal point; a floating-point constant (optional on some configurations) is composed of an optional sign, a mantissa of 1-16 digits, and an exponent. A non-numeric literal is composed of a string of characters enclosed in quotation marks; the maximum length varies with each configuration but is generally 120 characters.

Figurative constants are COBOL reserved words which cause the insertion of constants. Some figurative constants are: ZERO/ZEROS/ZEROES which insert 0 in the receiving field; SPACE/SPACES which insert spaces in non-numeric data items; BLANK WHEN ZERO which inserts spaces in a numeric field (this is specified in the data description definition in the DATA Division); ALL in combination with a character which will repeat a character or a character string when storing in the receiving field; LOW-VALUE/LOW-VALUES which inserts the lowest numeric value.

JOVIAL

There are five major categories of constants defined in JOVIAL; these are:

1. numeric constant
   - integer
   - floating
   - fixed
   - octal
2. dual
3. literal constant
   - hollerith
   - transmission code
   - octal
4. status constant
5. Boolean constant

The symbol <digits> is used in the following definitions to mean the decimal digits 0-9, and brackets [ ] indicate an optional choice.

Integer constants have the form:

$$[\pm] \langle \text{digits} \rangle \; [\text{E} \; \langle \text{digits} \rangle]$$

where E <digits> defines a power of 10 by which the first <digits> are multiplied.

Floating constants have the form:

$$[\pm] \left\{ \begin{array}{l} \langle \text{digits} \rangle \cdot [\langle \text{digits} \rangle] \\ \cdot \langle \text{digits} \rangle \end{array} \right\} \; [\text{E} \; [\pm] \; \langle \text{digits} \rangle]$$

where E [±] <digits> defines an optionally-signed power of 10.

Fixed constants have the form:

Floating constant A[±]<digits>

where A [±] <digits> represents a binary scaling factor defining the number of binary places to the right of the binary decimal point which are available for fractional values.

Octal constants have the form:

O (<octalnumber>)

where <octalnumber> is a string of digits which may assume the values 0 through 7.

Dual constants, which represent naturally-paired values, such as X and Y coordinates, have the form:

D (<constantvalue>, <constantvalue>)

where <constantvalue> can be an integer constant, fixed constant, or an octal constant -- but both <constantvalue> numbers must be the same type. Further, if the <constantvalue> pair is fixed, both values must have identical A [±] <digits> components.

Literal constants may be represented by two encoding schemes; Hollerith is the machine language representation and transmission is the alphanumeric representation.

Transmission constants have the form:

<digits> T (<characters>)

Hollerith constants have the form:

<digits>H (<characters>)

where in both cases <characters> is a string of any legal JOVIAL characters as well as any additional implementation-defined characters, and <digits> gives the number of characters (including spaces) between the parentheses.

Status constants which have values that are, in essence, mnemonic labels are of the form:

V (<identifier>)

where <identifier> is any legal JOVIAL identifier (see A.2) and represents a non-negative integer value.

Boolean constants have the form: $\left\{\begin{matrix}0\\1\end{matrix}\right\}$

where 0 means false, 1 means true, and differentiation from integer constants is by context.

PL/I

PL/I constants may be categorized as:

1. arithmetic constants
   - decimal fixed-point
   - binary fixed-point
   - sterling fixed-point
   - decimal floating-point
   - binary floating-point
   - imaginary

2. string constants
   character string
   bit string
3. label constants

A decimal fixed-point constant consists of one or more decimal digits with an optional decimal point and optional sign. If the decimal point is absent, it is assumed to be immediately to the right of the right-most digit. The form is:

[±] [<digits>] [.] [<digits>]

Binary fixed-point constants consist of one or more binary digits, immediately followed by the letter B, with an optional binary point and preceding sign:

[±] [<binary digits>] [.] [<binary digits>]B

Sterling fixed-point data is written as pound, shilling, and pence fields, separated by periods and followed by an L (e.g., 0.4.6L).

Decimal floating-point constants contain decimal digits followed by the letter E, followed by an optionally signed decimal integer exponent. The constant may be signed and contain a decimal point. The form is:

[±] [<digits>] [.] [<digits>] E[±] [<digits>]

The binary floating-point constant is similar to the decimal floating-point constant except that it consists of

binary digits and the entire constant is terminated by a B. The form is:

[±] [<binary digits>] [.] [<binary digits>] E [±] [<digits>] B

An imaginary constant is written as a real constant of any type, except sterling fixed-point, immediately followed by the letter I (e.g., 27I, 2.46E-2I, 101.1BI). There are no complex constants in PL/I. The effect is obtained by writing a real constant and an imaginary constant.

A character string constant includes any digit, letter, or special character allowed by the machine implementation enclosed in single quotation marks. A bit string constant is a series of binary digits enclosed in single quotation marks and followed by a B (e.g., '10110'B). The use of a bit string of length 1 achieves the effect of a Boolean constant.

Statement-label constants are prefixed to statements and constitute that set of values which statement-label variables may be assigned.

A.4 Basic Elements - Operators

The nature of programming involves the manipulation of data and implies a set of operators to accomplish specific transformations. Operators may be classified as computational, relational, logical, data moving, string, and editing.

Programming languages usually have provisions for the basic arithmetic operators: +, -, *, / and **. Operators which require

two operands are referred to a dyadic, binary, or infix operators; operators requiring only one operand are referred to as monadic, unary, or prefix operators. Plus (+) and minus (-) are operators which have both a monadic and dyadic form. The rules of the programming language will specify the hierarchy of execution for expressions involving multiple operators.

The relational operators compare the specified operands to determine the validity of the indicated relations; such tests include $=$, $\neq$, $>$, $<$, $\geq$, $\leq$, $\not>$, $\not<$. Languages differ to the extent that these operators have unique symbols or are specified via the use of reserved words.

The logical operators, AND, OR, and NOT (logical conjunction logical disjunction, and logical negation) provide the capability to perform logical operations on Boolean variables, and may be combined in one statement to generate logic tests of great complexity.

The primary data movement operator is the = (i.e., assignment) and is basic to all languages. However, programming languages oriented toward data processing applications and report generation usually include more sophisticated data handling operators.

A string is defined as a connected sequence of symbols, either characters or bits. String operators provide the capabilities for manipulating this data type; representative of such operations are concatenation, decomposition or substringing, and pattern searching.

Editing operators allow for the validity checking of source data prior to being processed, and for the modification of the form and format of data prior to being output.

AED

The usual arithmetic operators (+, -, *, /, **) are permitted in REAL, INTEGER or mixed expressions, with type POINTER also allowed in certain cases where only + or - are used.

Relational operators apply between numbers (INTEGER, REAL or mixed) or between pointers, but not between a pointer and a number. These are: GRT, GEQ, EQL, NEQ, LEQ, and LES (greater than, greater than or equal, etc.).

The Boolean operators are AND, OR, NOT, and IMP (A implies B; i.e., (NOT A) OR B).

The primary data movement operator is = (assignment). A number of stack operators are also defined, but these have been little used and are not being implemented in newer compilers.

COBOL

In COBOL, operators may be expressed by a character(s) symbol or a reserved word verb.

Computational operators are:

+	addition	ADD
-	subtraction	SUBTRACT
*	multiplication	MULTIPLY
/	division	DIVIDE
**	exponentiation	
unary +		
unary -		

The hierarchy of execution is unary plus and/or minus, exponentiation, multiplication and/or division, addition and/or subtraction. COBOL has a special verb COMPUTE, which allows the combination of computational operators in an equation type format.

Relational operators are expressed in the following manner:

IS [NOT] GREATER THAN or IS [NOT] >
IS [NOT] LESS THAN or IS [NOT] <
IS [NOT] EQUAL TO or IS [NOT] =
EQUALS
EXCEEDS

Logical operators are OR, AND, OR NOT, and AND NOT.

Data movement is accomplished in COBOL through use of the = and reserved words. Special data movement is achieved through use of the REPORT Section definitions when data is being transferred for report writing; this movement will be discussed under report-writing procedures. The reserved words associated with data movement are:

MOVE	- moves data from one item to another.
MOVE CORR/CORRESPONDING	- moves data named similarly.
EQUALS (or =)	- moves manipulated expressions.
TO or GIVING	- specifies the receiving field of a computational result.

Character strings in COBOL may be operated on directly by the EXAMINE verb, whereby a string is examined TALLYING the number of occurrences of a specified character or character string and/or REPLACING a specified string with another string (which may be indicated by a constant or a named variable). Other options for this verb are: UNTIL FIRST occurrence of a string, ALL occurrences, and LEADING occurrence. Other character string operations may be achieved by overlaying a character string with a redefining structure, where different parts of the character string may be referenced directly by an elementary item name.

COBOL uses single characters or character combinations, specified through a data item picture clause, to edit data as it is input, output, or moved internally. The editing controls are:

B	blank or space
0	zero
+	plus
-	minus
CR	credit
DR	debit
Z	zero suppression
*	check protection
$	currency
,	comma (decimal point option)
.	period (decimal point)
L	variable length data item

Other editing controls are specified in the Report Section. These are described as special features of COBOL.

Standard picture specification characters are:

9	for numeric data representation
A	for alphabetic data representation
X	for alphanumeric data representation
V	for an assumed decimal point (in a numeric item only)

The 9, A, and X may not appear in the same picture clause defintion.

JOVIAL

Computational operators consist (in order of descending priority) of unary plus and minus, exponentiation (**), multiplication (*) and division (/), addition (+) and subtraction (-). There is also an absolute value operator consisting of the bracketing symbols (/ and /) and an ABS function.

Relational operators consist of EQ, GQ, GR, LQ, LS, and NQ.

Logical operators consist of AND, NOT, and OR.

There is a Boolean function, ODD (see C.1.b.6).

Data movement operators consist of value assignment (=) and value exchange (==).

There are four string operators provided as functions: BIT, BYTE, CHAR, MANT. Each of these substrings a portion of a value and returns the numeric equivalent (see C.1.b.6).

There are five table operators provided as table functions based on various table aspects: ALL, ENT, ENTRY, NENT, NWDSEN. The returned values of ALL, NENT, and NWDSEN are integer; ENT and ENTRY return an entire table entry value.

There is a file operator, the function POS, which returns an integer value representing a file's position, and which can also be used on the left of an assignment statement to change the position.

PL/I

In PL/I the computational operators are:

+	addition or prefix plus
-	subtraction or prefix minus
*	multiplication
/	division
**	exponentation

Relational, logical, or data movement operators can be applied to numeric, character string or bit string data. The relational operators are:

60 Character Set		48 Character Set
>	greater than	GT
¬>	not greater than	NG
>=	greater than or equal to	GE
=	equal to	=
¬=	not equal to	NE
<=	less than or equal to	LE
<	less than	LT
¬<	not less than	NL

Logical operators are:

60 Character Set		48 Character Set
¬	NOT	NOT
&	AND	AND
|	OR	OR

The assignment operator, =, constitutes the only data movement operator.

The string operator, which can be applied to either two bit strings or two character strings, is:

60 Character Set		48 Character Set
||	concatenation	CAT

PL/I also contains 13 built-in string handling functions. They are:

BIT	converts value to bit string
BOOL	result of Boolean operation on two bit strings
CHAR	converts value to a character string
HIGH	forms string from highest character in collating sequence
INDEX	starting location of specified substring
LENGTH	length of string

LOW	forms string from lowest character in collating sequence
REPEAT	forms new string by concatenation with itself
STRING	concatenates all elements into single string
SUBSTR	extracts substring of original string
TRANSLATE	converts substring of original string into new substrings
VERIFY	compares two strings
UNSPEC	returns bit string of given value

Editing is accomplished in PL/I via the Picture Specification Characters, which are used in a manner similar to COBOL's data item picture clause. The picture characters edit data both for edit-directed input and output, and in assignment statements. The editing specification characters are:

Character String Specification:

X	any character
A	alphabetic character or blank
9	decimal digit or blank

Numeric Specification:

9	decimal digit
V	assumed decimal point
Z	decimal digits with leading zeros replaced by blanks
*	decimal digits with leading zeros replaced by *
Y	decimal digits with all zeros replaced by blanks

Insertion Characters:

,	possible comma insertion
.	possible point insertion
/	possible slash insertion
B	blank
$	currency
S	sign (+ or -)
+	plus sign or blank (if < 0)

More specialized editing characters are found in Picture Specification Characters of IBM PL/I Reference Manual.

Table II represents a summarization of the relational operators provided by the subject languages.

Table III

Summarization of Relational Operators

Language / Algebraic Symbol	AED	COBOL		JOVIAL	PL/I	
		Reserved Word	Symbol		60 Char	48 Char
=	EQL	EQUALS	=	EQ	=	=
≠	NEQ	UNEQUAL TO	NOT =	NQ	¬=	NE
>	GRT	GREATER THAN	>	GR	>	GT
<	LES	LESS THAN	<	LS	<	LT
≥	GEQ			GQ	>=	GE
≤	LEQ			LQ	<=	LE
≯		NOT GREATER THAN	NOT >		¬>	NG
≮		NOT LESS THAN	NOT <		¬<	NL

A.5 Basic Elements - Key Words/Reserved Words

Key words are character strings which, when used in proper context, convey a specific meaning to the compiler. Reserved words exist in some languages and may not be used as identifiers (i.e., variables and labels). The use of reserved words may make programs more readable but at the expense of programmer convenience and language extendability. The addition of new key words into a language may cause conflicts with existing programs.

AED

AED employs some 117 reserved words. These, together with 15 operators and special punctuation items (e.g., $=$),

are recognized at the lexical level in the compiler, and cannot be used as general variable names. All but 56 of these begin or end with a '.', and future reserved words will follow this convention, so that avoiding the use of such words for variable names will preclude the possibility of future conflicts.

COBOL

There are a total of 231 reserved words in COBOL, which are categorized as:

1. Key words
2. Optional words
3. Connectives

Key words are required in a COBOL entry. There are three types: verbs (e.g., ADD, READ, ENTER, CALL), required words (e.g., TO, IS), and functional words with special meanings (e.g., ZERO, NEGATIVE, SECTION).

Optional words are included in an entry for readability. No penalty is involved if they are omitted, but a compilation error occurs if they are misspelled or replaced.

Connectives are of three types:

Qualifier	-	OF, IN
Series	-	a comma is used to connect a series of operands
Logical Connectives	-	AND, OR, AND NOT, OR NOT

JOVIAL

Key words and reserved words in JOVIAL consist of the following:

ABS	ALL	AND	ARRAY	ASSIGN	BEGIN
BIT	BYTE	CHAR	CLOSE	DEFINE	DIRECT
END	ENT	ENTRY	EQ	FILE	FOR
GOTO	GQ	GR	IF	IFEITH	INPUT
ITEM	JOVIAL	'LOC	LQ	LS	MANT
MODE	NENT	NOT	NQ	NWDSEN	ODD
OPEN	OR	ORIF	OUTPUT	OVERLAY	POS
PROC	'PROGRAM	RETURN	SHUT	START	STOP
STRING	SWITCH	TABLE	TERM	TEST	

Any future additions to the list must begin with a prime (') to prevent possible confusion with names or labels in existing programs. This provides downward compatability for future extensions to the JOVIAL language.

PL/I

Key words are identifiers (see A.2). A key word, which has special meaning to the compiler only when used in proper context, can specify such things as action to be taken, the nature of data, and the purpose of the name. Some key words can be abbreviated.

Key words are not reserved words in PL/I. Only when the 48-character set is being used are there any reserved words; they are the relational operators (GT, NG, GE, NE, LE, LT, and NL), the logical operators (AND, OR, NOT), and the string operator, CAT.

A.6 Punctuation

Punctuators serve as delimiters within a program and may take the form of either graphic symbols or key words. Some of the more common functions performed by such punctuators are to terminate statements, to separate items in a list, to delimit comments, and to delineate subexpressions within statements.

AED

AED punctuation includes:

$	-	used as a separator (as the ; in ALGOL) and after a label
(and)	-	used for conventional grouping
,	-	used to separate elements of a list
...	-	used to initiate a comment and
//	-	used to terminate a comment
$=$ and $/$	-	used in synonym defining

At least one blank may be required to separate adjoining alphanumeric items, such as variable names, key words, and constants, but blanks are otherwise significant only in character string literals.

COBOL

COBOL punctuation characters consist of the following:

Period	Every header, label, sentence, record description, and elementary data item description must end with a period.

Comma	A comma may be used to separate clauses and as a series connector.
Semicolon	May be used to separate statements in a data description entry or a sentence.
Space/Blank	May be used as a series connector. A space must be used between words and operators, and between words and words. Some implementations have special requirements concerning the presence of a space on either or both sides of the following special characters: + - ( ) < > = ; and period. Spaces may not be imbedded in a name or numeric constant. Spaces may be used to separate clauses and statements in a sentence.
Quote	Quotation marks are used to enclose a literal character string constant.
Parentheses	Left and right parentheses are used to delineate groupings, indices, and subscripts. They must always appear in a balanced set.
Asterisk	This symbol denotes a remark line, if it appears in the continuation column of the input 'card'.

JOVIAL

JOVIAL punctuation, in addition to key words and operators, consists of the following list:

blank	-	a basic language element separator
$	-	statement terminator
.	-	label terminator
...	-	special separator for input lists
,	-	member separator for lists (e.g., subscript and argument lists)
=	-	symbol for assignment or identification of output in argument lists
==	-	symbol for assignment-like exchange of values
(and)	-	argument list and subexpression delimiters
($ and $)	-	subscript delimiters
BEGIN and END	-	statement group or constant initialization list delimiters
JOVIAL and DIRECT	-	assembly language group delimitors
START and TERM	-	program delimiters

PL/I

Each PL/I statement must be terminated by a semicolon. A label or condition prefix is connected to statements by colons. Commas are used to separate items in arrays. Periods are used to qualify variables (e.g., A.B implies variable B in structure A). The character pair /* indicates the beginning of a comment and the same character pair reversed, */, indicates its end. Comments are permitted wherever blanks are allowed, except within data items (e.g., within a character string). Comments can be treated as blanks and therefore replace blanks.

Blanks may be used freely in a PL/I program. They may or may not surround operators and most other delimiters. One or more blanks must be used to separate identifiers and constants that are not separated by other delimiters. However, identifiers, constants (except character string), and composite operators (e.g., ¬ =) cannot contain blanks. In general, any number of blanks may appear wherever one is allowed.

B.1.a Data Types and Organization - Problem Data Types

The power and versatility of a programming language is directly related to the scope of data types which may be defined and operated on. Problem data is that data which is to be processed by a program, in contrast to program control data which is used to control the execution of the program. The problem data types defined by a language reflect the primary application areas for which the language was designed. Languages oriented toward scientific processing might provide for complex, double-precision, and formal data, in addition to numeric data types of more general utility, such as integer and floating-point. The requirement to manipulate logical entities would predicate the definition of Boolean and status data variables, while business data processing would require alphanumeric character data.

AED

Numeric Data. AED INTEGER data has an implementation-defined range (whatever fits in a computer word); fixed-point fractions or mixed numbers are not represented. REAL data has an implementation-defined range and precision (whatever is supported by the hardware). There is no way to declare double-precision or complex mode for either INTEGER or REAL type. For problems involving much complex or double-precision computation, a call to FORTRAN would be recommended.

Logical/Boolean Data. Logical, or Boolean, data items have one of the values TRUE or FALSE. Such variables can be set equal to Boolean or relational expressions, and tested by IF or WHILE. Unless packing is explicitly called for, each Boolean item occupies a whole word.

String Data. Character string data is handled 'beneath the surface' in AED -- i.e., through one of the data types INTEGER or POINTER (cf. A.3). This is similar in spirit to FORTRAN IV, where INTEGER type data may contain characters. Operations are generally handled by subroutine packages, the only concession made by the language itself being the ability to define character-string constants.

There are two basic internal forms: one-character-per-word right-justified (the SPRAYed form) and packed (or GLUEd). The first is convenient for handling as an INTEGER array (i.e., as individual characters). The second is the preferred form for handling the string as such. A POINTER acts as a handle; it points to an area comprising the string plus a standard header containing the length, a forward chain pointer, and other incidentals. Such a string may be extended by chaining another onto it.

The collating sequence is implementation-defined. Truncation, justification, and filler conventions are generally not defined in the language, as not being applicable to the storage method. The same is true of the concept of 'varying' string.

Bit strings are limited to strings one computer word in length, and are formally declared as INTEGERs. Such strings may be shifted or combined in parallel Boolean operations (AND, OR, and exclusive-OR). Again, this is close to the facility available in some FORTRAN IVs.

Status Data. Status data in the sense of variables having designatory (non-quantitative) numeric values (e.g., 1 denotes 'passenger auto', 2 denotes 'bus', 3 'trailer', etc.) are handled as type INTEGER in AED, using the SYNONYM (or possibly the MACRO) capability to define the number-meaning correspondence. It would be possible to write:

```
CLASS = BUS
```

or

```
IF CLASS NEQ PASS.AUTO . . .
```

COBOL

Numeric Data. Problem data is described by the use of a PICTURE/PIC clause with optional USAGE (i.e., COMPUTATIONAL, DISPLAY, INDEX, or COMPUTATIONAL-n), SYNCHRONIZED, and JUSTIFIED clauses. Synchronization of numeric data may be either to the left or to the right, thereby causing word alignment (where word alignment is a function of the implementation restrictions) and the introduction of slack positions between the data elements.

Valid characters for numeric data definition in editing clauses are 9, V, P, and S, where 9 indicates a number, V an assumed decimal point location, P the scale factor, and S the optional sign. Floating-point pictures will include E to denote the exponent.

Fixed-point data is composed of 1-18 digits, an assumed decimal point, and an optional sign. This data type may be defined with USAGE COMPUTATIONAL or USAGE DISPLAY. Integer data is similar to fixed point with no assumed decimal point imbedded (i.e., rightmost position is the assumed decimal point). Complex data is not directly available as

part of the COBOL language. Floating-point is indicated, when this option is available, by COMPUTATIONAL-n, where n is a configuration/implementation-defined number (e.g., IBM 360/370 version uses COMPUTATIONAL-1 for binary single-precision floating-point and COMPUTATIONAL-2 for binary double-precision floating-point).

It should be noted that COMPUTATIONAL generally indicates 'internal binary' format and DISPLAY indicates 'external decimal' format. There is a third option available on some configurations (expressed as a computational variation) which indicates 'internal decimal', which is comparable to a packed internal decimal format.

Implementation of COMPUTATIONAL (COMP) -n and DISPLAY-n allows different problem data types on different computers. The -n portion of the USAGE descriptor is sequentially assigned to meet the manufacturer's data availability specifications. It would be through this USAGE clause that bit manipulation might be achieved, and any other data specifications a manufacturer might wish to include as part of a compiler.

Logical/Boolean Data. In COBOL, the logical (Boolean) data type is not directly part of the language. This may be implemented on some machines by use of a form of USAGE COMPUTATIONAL-n or USAGE DISPLAY-n for single bit expressions. Generally, the program would have to either CALL another routine or ENTER an assembly language routine to manipulate or set up logical data of the one-bit type.

Status switches, defined in the ENVIRONMENT Division, and conditional variables, defined in the DATA Division, may be used as a type of logical data in that they can be coded to assume only two values ON or OFF, or zero (0) or one (1), respectively.

String Data. A character string may be composed of from 1 to a machine-specified upper limit of alphanumeric characters. By use of the OCCURS DEPENDING option, a character string may be varying in length. In the definition of an elementary item character string, a JUSTIFIED clause may be used. If this option is used on a receiving field, truncation will take place on the left of a sending field if the field is too large, otherwise normal truncation is on the right. Unused character positions, under all options, are filled with blanks/spaces. Justification overrides normal character string positioning and right-justifies. The data character string constants must be enclosed in quotes. Character strings may be used as values for conditional variables.

Bit strings are not a basic part of the COBOL language. Some implementations may provide for this type of data by use of the DISPLAY or COMPUTATIONAL verb number options.

Status Data. Status data may be defined in two program divisions, ENVIRONMENT and DATA. Those data items described in the former are classified as SPECIAL-NAMES and serve as a program interface with the implemented system. Some of the items usually have an ON or OFF status; others may have time of day, date, next job name, next task name, etc.

Status data described in the DATA Division are the condition names associated with conditional variables. These names must immediately follow the condition variables for which they represent either a single value, a set of values, or a range of values. There are no restrictions on the data type of condition name values, except that the value specified must conform to the picture clause specified for the conditional value.

```
Example:  01 MARITAL-STATUS    PICTURE (A)
             88  SINGLE        VALUE IS 'S'
             88  MARRIED       VALUE IS 'M'
             88  WIDOWED       VALUE IS 'W'
```

JOVIAL

Numeric Data. JOVIAL provides for five categories of numeric data: integer, floating, fixed, octal, and dual. Integer data can include constants, variables, and results of functions. Functions providing integer values consist of BIT, CHAR, 'LOC, POS, and NENT. Floating data can include constants and variables. There are two functions, CHAR and MANT, which operate on floating data. Fixed data can include constants, variables, and the MANT function. Octal data is provided only as constants. Dual data is provided as an ordered pair of integer or fixed (but not floating) constants or variables. JOVIAL defines dual data as a type distinct from numeric, literal, etc. Complex data is not available as such.

Logical/Boolean Data. JOVIAL provides for Boolean problem data as constants, variables, and the result of the ODD function and relational operators.

String Data. JOVIAL provides for two types of string problem data: transmission code and hollerith. Transmission code is language-dependent and hollerith is implementation-dependent; both types may be used as constants or variables. The BYTE function provides substring values of both data types.

The equivalent of bit substring functions are the BIT and CHAR functions which return integer values, and the MANT function which returns a fixed value.

Strings are fixed-length only, and are padded with leading blanks to adjust lengths (e.g., in relational expressions).

Status Data. JOVIAL provides a type of data known as status, which is used in logical processing and represented as constants or variables. Status data represents logical states and identifiers as a dense set of non-negative integer values, so that, for aircraft, the status constant V(MAINT) might correspond to an aircraft being in maintenance and might be represented by the value 0. Other aircraft statuses might be V(AIRBRN), V(LOADING), and V(TAXI) and might be represented by the values 1, 2 and 3, respectively. Since the programmer has complete control of the mnemonics, readability of code can be improved considerably without the sacrifice of storage space, which would occur if status values were stored as literal rather than integer values.

PL/I

Numeric Data. There are three types of fixed-point data: decimal, sterling, and binary. The maximum number of decimal digits allowed for IBM System/370 or 360 implementations is 15. (Although PL/I compilers now exist on other machines, the majority of PL/I users run on IBM systems.) The default precision for decimal fixed data is five decimal digits, all to the left of the decimal point. The internal coded arithmetic form of decimal fixed-point data is packed decimal.

Sterling fixed-point data consists of three digits separated by periods, followed directly by the letter L (e.g., 3.10.8L). All three fields (pounds, shillings, and pence) are required. Sterling fixed data is maintained internally as a decimal fixed-point number representing the equivalence in pence. The maximum number of digits allowed in the pounds field is 13. The pence field may contain an optional decimal point (e.g., 3.10.8.5L). The integer part must be less than 12 and the fractional part less than or equal to 13 minus the number of digits in the pounds field. Sterling fixed-point data may be deleted from PL/I in the near future.

The maximum number of binary digits in binary fixed-point data is one full computer word, the first bit of which is the sign bit. An identifier with no declaration is assumed to be a binary fixed-point variable, if its first letter is I through N.

There are two types of floating-point data: decimal and binary. The maximum precision on IBM 360/370 systems is 16 digits and the exponent cannot exceed two digits.

Values may range from approximately 10^{-78} to 10^{75}. The default precision is six decimal digits. If an item is assigned to a variable with a smaller declared precision, then truncation may occur on the right (i.e., least significant bits are truncated).

The internal code is normalized hexadecimal floating-point, with the decimal point to the left of the first digit. An identifier with no declaration is assumed to be decimal floating-point, if its first letter is A-H, O-Z, $, #, @.

The maximum precision allowed for binary floating-point data on an IBM machine is 53 binary bits and the range is approximately 2^{-260} to 2^{252}. The internal code is normalized hexadecimal floating-point. The default precision is 21 binary bits. The exponent cannot exceed three decimal digits.

In PL/I, integer data is expressed by declaring the variable to be fixed-point, with no precision to the right of the decimal point (e.g., FIXED (6,0) or simply, FIXED (6)).

An imaginary constant is written as a real constant immediately followed by the letter I (e.g., 3.96I). A complex constant is written as a real constant combined with an imaginary constant and is of the form:

real constant $\left|\pm\right|$ imaginary constant

(e.g., 27-38I).

Logical/Boolean Data. In PL/I, logical data is a bit string of length one.

String Data. A PL/I character string can include any digit, letter, or special character recognized as a character by the particular machine configuration. A character-string constant must be enclosed in single quotation marks. A null string constant is written as two quotation marks with no intervening blanks. If a string is longer than the length declared for the variable, it is truncated on the right; if shorter, it is padded on the right with blanks. Character strings may be declared to be of varying lengths. The length attribute of the variable is then the length of the data item most recently assigned to it.

Internally, each character of the string occupies one byte of storage. The maximum length for character string variables is 32,767 bytes. The maximum length for character-string constants varies according to the compiler, but it never will be less than 1,007. The minimum length of a character string is zero.

The minimum length for a bit-string variable is 32,767 on the IBM F compiler, but the length varies with other compilers. The maximum length for a bit-string constant depends upon the amount of storage available to the compiler, but never will be less than 8,056. Bit strings, like character strings, may be declared to be of varying length. A null bit string consists of two single quotation marks followed by the letter B (i.e., ''B). If a string is longer

than the length declared for the variable, the rightmost digits are truncated; if shorter, padding is on the right with zeros.

Status Data. PL/I does not contain status data. The same effect can be achieved by the use of pre-processor macros.

B.1.b Data Types and Organization - Program Control Data

In addition to the problem data on which the program algorithms operate, some higher-level programming languages provide for program control data, whereby such quantities as labels, procedure names, pointers, and event data may be manipulated to affect the execution of the program. While operations on such data types are generally limited, they do provide the mechanisms to access machine addresses and to dynamically control the processing sequence.

All but the most elementary programming languages provide for the naming of program units via statement labels and procedure names; however, the power of the language is greatly enchanced if, in addition to providing a simple branching capability, these program-unit labels may themselves be treated as data (e.g., as operands in assignment statements or as arguments in parameter lists.)

Access to storage addresses is provided via pointer and offset data types; a pointer is a locator variable which defines an absolute core location, while an offset defines a core location relative to a specified area.

Languages in support of multiprogramming computer systems provide for asynchronous program operation by the definition of such

data types as task (a named function module capable of parallel execution with a given priority) and event (a communication/synchronization variable which indicates the completion status of an identifiable point within a task).

AED

Label/Switch Control Data. Statements may be labeled and thus be the target of GOTO statements. In addition, the value of LOC (<label>) is a pointer to the program point -- in effect, a label variable. An indirect branch may be made to the statement, even from a descendant procedure several generations removed, via the DOIT function.

A switch is, in effect, an array of labels. An element of the array is selected by the GOTO (e.g., GOTO SW(I), where I is of type INTEGER, will cause a branch to the Ith label associated with SW).

Procedure Control Data. Procedures are named items whose 'values', so to speak, are fixed at compile time. That is, the procedure cannot be modified dynamically in the sense of an interpreted LISP procedure. However, the name, or more properly the location, can be handled by the POINTER data type. As with labels,

<procedure-ptr> = LOC (<procedure-name>)

yields a pointer that may be used as a handle for the program point; the function reference

DOIT (<procedure-ptr> [,<arg1> , <arg2>...])

effects an indirect call.

Pointer/Offset Control Data. Pointer data is a basic element of AED. Using pointers as links, data structures of

any complexity can be built. Offsets as such are not provided, although arithmetic manipulation of pointers may be used to achieve the same purpose.

Multitasking Control Data. Event data, in the PL/I sense of a synchronization mechanism between separate tasks, is not present in AED because multitasking is not supported.

COBOL

Label/Switch Control Data. Labels are specified for a statement only if the statement is to constitute a paragraph. In COBOL, labels are used for divisions, sections, and paragraphs. There are some reserved words used for section and paragraph labels; these are usually in the IDENTIFICATION, ENVIRONMENT, and DATA Divisions. Paragraphs in the PROCEDURE Division must be uniquely named and conform to the rules of a word. Paragraph names may be numeric. Sections must be uniquely named (since there is no method of qualification available at a higher level of reference) and followed by a space, the reserved word SECTION, and a period.

Labels are treated as data in the sense of being able to change the address of a label via the ALTER verb. If a paragraph is composed of a single GO TO statement, this may be changed by:

ALTER [<para-label-1>] TO [<para-label-2>]

There is a conditional clause available for the ALTER verb:

ALTER (---) DEPENDING ON (---).

Switches as a type of control data are not available in Standard COBOL. Paragraph labels are used for all transfer destinations based on calculations and conditional and unconditional testing statements.

Procedure Control Data. Procedure names conform to word restrictions for paragraph, section, and program identification. Internal procedures may be sections, a group of paragraphs, or a single paragraph. External procedures are full COBOL programs or programs coded in any other language with the proper COBOL interface set up for data transfer. Reference is made to an internal procedure by use of the PERFORM verb; reference is made to an external procedure by use of the CALL verb.

An internal procedure may be performed as a procedure when called out of sequence or as a paragraph when entered in a normal sequential statement execution process.

Pointer/Offset Control Data. Pointer and offset data are not part of the COBOL language. The language does provide a capability for Table Handling by using subscripts or indices. An index, declared in the DATA Division, may be modified only by a SET, SEARCH, or PERFORM statement. The index is defined by a USAGE IS INDEX clause. Relative indexing may be achieved by following the index name with + or - and a numeric literal. Subscripting and indexing may not be used together in a single reference. Index values are not defined in the program since the format and allocation are dependent on the system. The value represents a displacement from the beginning of a table and is coded in binary.

Multitasking Control Data. Event data is not presently part of the COBOL language.

JOVIAL

Label/Switch/Control Data. JOVIAL does not recognize labels as data types. Statements may be labeled, however, by prefixing the statement with a unique identifier followed by a period; multiple labels (each an identifier followed by a period) may be associated with the same statement.

Procedure Control Data. JOVIAL does not recognize procedures as data types. Names of CLOSEd procedures, however, can be passed as members of an input parameter list (see C.1.b.7). Locations of external procedures ('PROGRAMs) are available as integer data with the 'LOC function.

Pointer/Offset Control Data. JOVIAL does not recognize pointers as data types. Locations of variables and external procedures ('PROGRAM) are available as integer data with the 'LOC function. These values represent absolute core locations; JOVIAL is not oriented towards dynamically-relocatable environments.

Multitasking Control Data. JOVIAL makes no provision for multitasking data.

PL/I

Label/Switch Control Data. A label data item is either a label constant or the value of a label variable. A label constant is an identifier written as a prefix to a statement and connected to the statement by a colon (e.g., A: B=C). The statement can be executed either by normal sequential execution or by transfer of control to this statement by means of a GO TO statement. Label variables are variables

that take on the value of label constants, so that any reference to the variable would be the same as a reference to the label constant.

Switches do not exist in PL/I but the same effect can be achieved by using label arrays.

Procedure Control Data. Every procedure statement must have a label which is the procedure name. It is the primary point of entry through which control can be transferred to the procedure, but not the only one. The name of a procedure can be passed as data in a PL/I program. Therefore, one procedure can call an entry variable where the name of the entry point is passed as data.

Pointer/Offset Control Data. The value of a pointer variable is effectively an address of a location in storage, and so it can be used to qualify a reference to a variable that may have been allocated storage simultaneously in several different locations, all of which exist concurrently.

Offset variables specify a location relative to the start of a reserved area of storage and remain valid when the address of the area itself changes. This reserved area is another type of PL/I data and it can be assigned or transmitted complete with its contained allocations.

Multitasking Control Data. Event variables are used to coordinate the concurrent execution of a number of procedures, or to allow a degree of overlap between a record-oriented input/output operation and the execution of other

statements in the procedure that initiated the operation. Task data is used to control relative priorities of different tasks when multitasking.

B.2 Data Types and Organization - Logical Data Organization

The nature of data processing is such that the restriction of data variables to scalar quantities would be totally inadequate; constructs are necessary to deal with organized aggregates of data, consisting of both homogeneous and non-homogeneous elements. To this end, higher-level programming languages usually provide for the definition of arrays and structures.

An array, consistent with mathematical notation, consists of an aggregate of data with the same attributes. In its simplest form, an array is a linear sequence of data items (i.e., a vector), but may take the form of a rectangular collection of elements (i.e., a matrix) and, ultimately, an n-dimensional matrix. The extent of the matrix dimensions allowed by a programming language will be a limiting factor governing the organization of problem data. References to elements within an array are via subscripts, which may be limited to constants or may be expressed as variables or expressions. Additional convenience results when negative subscripts are allowed, and when a succinct notation is provided whereby all the elements in a specified dimension may be accessed.

A structure, in contrast to an array, is a data aggregate comprised of data elements with differing attributes. This type of mechanism allows the intrinsic relationships between elements of data to be retained within the computer, thereby facilitating the information processing and retrieval. The data may be organized in a hierarchical manner and names may be associated with both major

units and ancillary subunits. Each language specifies rules for qualification of names, whereby ambiguities, resulting from the same data name existing in multiple hierarchies, may be resolved.

Arrays and structures may be combined to provide for logically-complex organizations of data. In particular, it is possible to have an array of structures, in which each element of the array is an entire structure; or conversely, a structure whose elements consist of arrays.

Lists are data structures which provide the facility for processing quantities of unstructured data whose storage requirements and logical relationships are transitory. As a result, lists need not be restricted to the data items themselves but, more typically, consist of locator variables which point to core locations and are used to chain data together in linked lists. Lists may be one-way (each element points to its successor), two-way (each element points to its successor and predecessor), and ring (the terminal element points to the head of the list).

Special types of lists consist of stacks (a push-down store or LIFO list), queues (FIFO lists), and trees (data elements with multiple successor elements).

AED

Arrays. One-dimensional arrays can be declared for any elementary data type in AED and referenced in the usual way. One unfortunate aspect of the AED notation is that such arrays cannot be components of beads. (A bead is a cell, or contiguous block of data accessed by pointer, corresponding to a BASED array in PL/I.)

Arrays of two of more dimensions cannot be declared in AED. As with double-precision or complex data, a call to FORTRAN is recommended. Also, the effect of multi-dimension arrays can be achieved through the macro preprocessor at the cost of considerable labor and compilation time.

Structures/Tables. Data may be collected into contiguous blocks, called 'beads', of mutually related items. Such beads may be linked together by pointers to form trees, lists, stacks, rings, or any arbitrary data structure.

List Structures. AED has a STACK data type, with seventeen associated operators for controlling data movement in and out. However, this type has fallen into disuse (no doubt partly because of the poor notation for operators), and will probably be omitted in future implementations. In general, data structures of any type or complexity may be devised on a 'do-it-yourself' basis using pointers.

COBOL

Arrays. Arrays in COBOL may be 1, 2 or 3-dimensional. They are indicated by the OCCURS clause in the data entry description given in the DATA Division. A variable-length array is indicated by the DEPENDING ON clause, which may be appended to the OCCURS clause. If the array is to be indexed, a KEY and sequence of the key (ASCENDING/DESCENDING) may also be specified, in addition to the INDEXED BY clause which specifies the index names to be associated with the array(s).

Any data type may be specified for an array. Variable length items may not be specified as arrays. VALUE clauses

may not be used with an OCCURS clause and care must be taken when using the REDEFINES clause, for implementations differ on the action taken if the storage size of the data descriptions do not match exactly.

Arrays are referenced in the PROCEDURE Division by using subscripts or index names. Subscripts must be either a non-subscripted data name or a numeric literal. Subscripts must not be zero or negative.

Structures/Tables. The majority of the data in COBOL is described in a record/table/structure form. All data items, except for items in arrays, are described with a level number, a name, and optional description clauses. There are two level numbers for non-continguous data: 01 and 77. The latter is used for a single data item and may only be used in the non-file sections of the DATA Division. Level number 01 indicates the beginning of a 'record', either actual I/O or internal data in a pseudo-record format (contiguous data). Level numbers determine hierarchy of data items within a table. Level number 01 is the highest level number and level number 49 is the lowest level. Elementary items in structures/tables may have similar names as long as each item may be uniquely qualified; that is, some higher-level name is different. Reserved words used for qualification are OF and IN.

An example of a structure in COBOL:

```
01  EXAMPLE-1.

    02  FIELD-2  OCCURS  3 TIMES.

        04 F-1   PICTURE A(8).
        04 F-2   PICTURE 999V99.

    02  FIELD-3      PICTURE  99  VALUE IS 0.

    03  FIELD-4      OCCURS   10   TIMES DEPENDING
                      ON FIELD-3.

        05 F-1       PICTURE  A(8).
        05 F-2       PICTURE  9V99.
```

As the example indicates, all level numbers do not have to be sequential; they must, however, be ascending to indicate a lower level of subdivision. In the procedure referencing F-1, items would require qualification as: F-1 IN FIELD-4 [OF EXAMPLE-1], where the last part is optional unless there is another structure in FIELD-4 as an entry name with an F-1 subelement.

List Structures. Stacks, queues, lists, and trees are not specific language data types. This type of data organization could be achieved through arrays and structuring, as previously indicated.

JOVIAL

Arrays. There are two basic ways of defining arrays in JOVIAL. The TABLE statement provides for single-dimensioned arrays (vectors) of the table's elements -- in effect, an

array of structures. The length of the TABLE array may be rigid (fixed-length) or variable (varying-length). Storage allocation depends on whether the table is serial or parallel (see Structures/Tables).

The ARRAY statement provides for multi-dimensional arrays of a single element. ARRAY arrays are fixed-length only, and the maximum number of dimensions is implementation-defined. Storage is allocated starting with the leftmost dimension.

A third form of array is provided by the STRING statement, which defines, within TABLEs, multi-valued items; single-valued items use the ITEM statement instead. A dimensioned table which includes STRING elements in effect provides for two-dimensioned elements of structures; the leftmost dimension corresponds to the STRING and the rightmost to the TABLE.

Any data type may be arrayed as described above. Dimension values start at 0 (probably a holdover from assembler experience) and increase positively in steps of 1. Any expression may be used as a subscript.

Structures/Tables. Structures are provided by TABLE as any collection of data types. Repeating groups (defined by STRING) may be elements of a structure, and any one table may be further set up as a single-dimensioned array providing repetitions of the basic structure. Structured data may be packed and data names are not qualified by the structure name.

Two basic structure types are provided: serial and parallel. In serial tables, storage for all elements of an item are assigned sequentially in core until the table's dimension is satisfied, before storage for the next item begins. In parallel table, storage for each item of an entry (i.e., a single value of the table's dimension) is assigned sequentially in core until all items of the entry are allocated, before storage for the next entry begins. If a table is variable length, only serial structure is allowed.

List Structures. Stacks, queues, lists, and trees could be provided by the judicious application of structures/arrays and the appropriate data types, but none is explicitly available in the language.

PL/I

Arrays. All PL/I arrays are n-dimensional so that vectors and matrices can be represented by appropriately-dimensioned arrays. PL/I also supplies library functions to manipulate these vectors and matrices.

A PL/I array is a collection of elements all of which have identical attributes. If the lower bound for an array is not specified, it is assumed to be 1. If the lower bound of an array is not 1, both upper and lower bounds must be stated explicitly, with the two numbers connected by a colon (e.g., DECLARE A (-4:11);). Data is assigned to arrays and stored internally in row major order (i.e., rightmost subscript varying most rapidly). The IBM F compiler allows as many as 32 dimensions for an array (machine-dependent). PL/I allows arrays of arithmetic data, strings, and labels. Any expression that yields an

arithmetic value can be used as a subscript. Cross-sections of arrays may be referred to by substituting an asterisk for a subscript (e.g., A(*,1) implies A((lower bound),1), A(lower bound)+1,1)...A((upper bound),1)).

Structures/Tables. In PL/I, a structure is a hierarchical collection of names. At the bottom of the hierarchy is a collection of elements; at the top is the structure name, which represents the entire collection of element variables (e.g., DECLARE 1 A

```
          2 B
            3 C
            3 D
          2 E
            3 C
            3 D
              4 C).
```

Elements of the structure need not have identical characteristics. Each element can be referred to separately by the use of qualified names. In a qualified name, high levels are used to qualify lower levels and each level name is separated by a period (e.g., A.B.C. or A.E.D.C). The maximum level number permitted is 255, but the true level number is 63 (since level numbers may be skipped) on IBM machines. Any level of a structure may be an array.

List Structures. Stacks may be built in PL/I by use of data with the attribute CONTROLLED. Stacking occurs when this variable is specified in an ALLOCATE statement. Any reference to the variable always refers to the most recent allocation. A FREE statement will cause the top of the

stack to be 'freed' and the stack 'pops-up'. The most recent previous allocation is once again available and its value becomes the value of the variable.

By using structures containing pointers, the user can construct any type of list, ring, stack, or tree. Pointers provide one of the most powerful capabilities of the PL/I language.

B.3 Data Types and Organization - Physical Data Organization

In addition to the logical data organizations described in the previous section, the physical organization of data within memory is often a consideration of concern to programmers. When the total storage requirements for a program exceed the available memory, compaction techniques must be employed. These include having multiple data items stored with a single computer word (i.e., packing) or having multiple data items share the same storage location at different times during program execution (i.e., overlaying).

To allow programmers to effect a trade-off between economy of storage and speed of access, which are both functions of the positioning of data elements within core, some programming languages allow for the specification of data storage relative to the physical characteristics of the media (e.g., justification within a word, or alignment on byte, word, or double-word boundaries).

A record is defined as a collection of related items of data, treated as a unit; records are comprised of fields and collections of records form files. Files constitute a basic data aggregate for large-

scale business and data management applications. A programming language may be considered adequate for these application areas to the extent that it provides the mechanisms to do file manipulations and to maintain the logical correspondence between the data as it exists on the external media and within the computer proper.

AED

Packing. Packing is specified in AED by an explicit PACK declaration, setting up the position and number of bits an item is to occupy within the machine word. The implied shifting and masking operations are compiled in for every reference to the item.

Alignment. Alignment is generally forced on word boundaries, with no choice except as provided by the PACK instruction.

Overlays. AED has no explicit provision for overlays. However, in the declaration of items making up a bead, giving the same offset (from the base pointer) to two or more items will have the effect of overlaying them. Dynamic overlaying may be achieved by setting pointers, as in PL/I.

Records. AED, having no I/O, does not distinguish records from any other data aggregate at the language level.

COBOL

Packing. Data packing is accomplished in COBOL in different ways. Generally, data storage is considered to be either contiguous or non-contiguous. The language does not require word alignment for data items; however, the user may force word positioning and data positioning by the attributes specified. Actual numeric data values may be stored in an 'internal decimal' form by using the COMPUTATIONAL-n option. This would, for example, strip zone bits from the numeric

data and pack two digits to a byte as on the IBM 360/370 series computers. Alphanumeric and alphabetic data may be positioned to the left or to the right in a data entry by using the JUSTIFIED clause. Normal character positioning is to the left. All truncation rules will be reversed when JUSTIFIED RIGHT is specified for the receiving data element. Justification will not cause any changes in physical data storage but COMPUTATIONAL-n will, if the data elements so described are in an array or a table/structure. Justification may only be specified for elementary items and may not be specified for level 66 (RENAMES) or 88 (condition names).

Alignment. Proper boundary alignment is not a requirement of the COBOL language. Implementation may cause some alignment at compile time; this usually consists of forcing level 01 data entries to begin on a 'computer word' boundary. The language provides a method of data alignment through the SYNCHRONIZED/SYNC [RIGHT/LEFT] clause. This clause is specified for elementary data items, usually numeric.

Proper boundary alignment is also a function of the type of data usage. COMPUTATIONAL and COMPUTATIONAL-n implementation determines the boundary alignment, whether half-word, full-word, or double-word. Therefore, the redefinition capability should be approached with care when redefining data whose boundary alignments may be affected by the presence of realigned data elements.

The appearance of either of these clauses (SYNC or COMP) may effect the storage and alignment of the remainder of the structure or record.

Overlays. Overlaying of data is achieved in three ways in COBOL: by the SAME AREA clause in the ENVIRONMENT Division I-O-CONTROL Section; by the multiple-record description (level number 01 being repeated) in the FILE Section of the DATA Division; and by the REDEFINES clause in all other non-file sections of the DATA Division.

The SAME AREA clause indicates the names of the files that are to occupy the same storage space during execution; only one file may be opened at a time. The multiple-record descriptions allow for the defining of several record formats for a file described with a FD, SD (sort), RD (report), or SA (saved area) descriptor; record lengths must be identical for each record in a file. The REDEFINES clause must immediately follow the data being redefined. The scope of the redefinition is governed by level numbers. It starts with the data name and ends when a level number less than or equal to that of the data name being redefined is encountered. The format is:
<level-number> <data-name-1> REDEFINES <data-name-2>.

Another overlay capability does not affect storage but provides the capability to rename, thereby allowing alternate groupings and possible overlapping of groups. More than one RENAMES clause may be specified for a record/structure. Level number 66 is reserved for this entry. All RENAMES must immediately follow the last data description entry for the record/structure associated with the entry. The format is:
66 <data-name-1>/RENAMES <data-name-2> [THRU <data-name-3>].

Records. The basic format of data in the COBOL language is the record in a 'file'. All record data is contiguous data and may be referenced by the 01 level number data name (comparable to a record name) or the subentry names. It is within the internal 'record' structure that all contiguous data is described for input and output files, report files, and sort files, as well as working-storage data descriptions. Data that are described only in an output-file description do not exist in user-available memory after the record has been written out. The REPORT Section allows for control information to be incorporated as part of the record description. More than one record format may be described immediately after the file description. All files must be described in the FILE Section; however, the detailed description of the file may be given in the WORKING-STORAGE Section and a single entry given in the FILE Section with the record name. This setup would allow for record concatenation during execution, if several records are required in memory at the same time.

All records for a file must have the same length. The maximum number of record descriptions is an implementation restriction.

JOVIAL

Packing. Packing applies only to items organized into tables or arrays and is provided at three levels: none, medium, and dense. No packing means that each item is allocated the least number of adjacent computer words which will hold it; medium packing means that each item is allocated the least number of adjacent bytes; and dense packing operates at the bit level. Only floating type data may not be packed at all, and literal items may be packed only to the byte level.

Alignment. JOVIAL does not provide explicit control of data alignment.

Overlays. Programs and data can both be overlayed so as to map either into identical storage locations or else into adjacent storage locations.

Records. Provisions for the physical arrangement of data in records are identical to those of single variables, arrays, tables, and entries.

PL/I

Packing. Control over the internal packing of variables can be accomplished in PL/I by the declaration of the variable. If a variable is declared to be binary-fixed, it is stored in a binary format with the first bit used as the sign bit. If the variable is declared to be decimal-fixed, it is stored in packed decimal format and if declared with a picture attribute, the variable is stored internally in a zoned decimal format.

Alignment. Most variables in PL/I can be declared to be either ALIGNED or UNALIGNED. ALIGNED data is aligned on storage boundaries corresponding to its data type requirements (i.e., full- and double-word items on word boundaries, half-word items on half-word boundaries, and character and bit strings on byte boundaries). UNALIGNED data is stored contiguously with the data element preceding it, and a word or double word is mapped into the next available byte boundary. Pointer, offset, label, event, and area data cannot be unaligned. The default for bit-string, character-string, and numeric data is UNALIGNED; all other data types default to ALIGNED.

Overlays. PL/I allows the user to overlay storage in either of two methods. The first is by using the DEFINED attribute when declaring a variable. This allows the user to specify that the variable is to represent part or all of the same storage as that assigned to other data. This attribute can be used for element, array, or structure variables.

The second method is to declare the variable BASED and position the pointer so that the variable shares the same position in core with part or all of another variable.

Records. PL/I records are comprised of single units; i.e., variables, arrays, structures, etc. Sections of data aggregates can be used as records only by overlaying a single unit over them.

C.1.a.1 Program Structure - Non-executable Units - Comments

Comments are in the nature of explanatory text, inserted by the programmer to clarify the processing being done and to supply the first level of program documentation. Since comments are merely passed by the compiler, they must be delimited from statements which require compiler action. Each language has its unique delimiters and rules regarding the allowable symbols and the placement of comments.

AED

Comments may be written virtually anywhere in the source program, bracketed by ... and // delimiters. In addition, there is an ALGOL COMMENT statement.

COBOL

The language has two methods of expressing comments and remarks. The first to be described is in ANS COBOL, but is not in CODASYL COBOL. It is anticipated that the asterisk notation will be the official standard for comments.

The first method uses two different reserved words. REMARKS is a paragraph label in the IDENTIFICATION Division only. Any sentence or statement is allowable as long as the physical format complies with the rules of COBOL with respect to indentation and punctuation. NOTE is a paragraph label in the PROCEDURE Division that also allows any sentence, statement, or combination of same, as long as all formatting rules of COBOL are obeyed.

The newest method specifies line comments which may be used any place in a COBOL program. A line comment is indicated by an asterisk (*) in the continuation position

of an input card. Each line of comment must have the *. A special variation is the stroke (/) in the continuation position. This character will cause the logical printer to eject to the top of the page prior to printing the line comment.

JOVIAL — Comments are introduced with a double prime ('') and terminated identically. In general, a comment may appear wherever a space is legal outside of literals.

PL/I — Comments are permitted wherever blanks are allowed, except within data items (i.e., in a character string), and are treated as blanks. The character pair /* indicates the beginning of a comment, the same pair reversed */ indicates the end of the comment (e.g., /* character string */).

C.1.a.2 Program Structure - Non-executable Units - Declaration/Specification Statements

Declarations, unlike statements, do not result in the direct generation of machine code, but rather supply the compiler with information relative to the structure of the program, the attributes of the data to be processed, and the interaction with the environment. Such information is necessary in order that proper and efficient code may be generated, sufficient space allocated for the storage of data structures, and the appropriate program linkages established. The format of such declarations is language-dependent, but a more fundamental difference derives from the extent to which these declarations must be explicated and are invariant. In particular, some languages require that all declarations be explicit, while others provide for implied declarations via default options and contextual

definitions. The binding of attributes to data is usually static and accomplished at compile time; however, some languages provide declaratives for the dynamic creation and deletion of data and delay such binding until execution time.

The multiplicity of data types requires that each data element be unambiguously defined in terms of attributes, which include such specifics as type (e.g., integer, floating-point, character-string, pointer), precision or length, dimensions and bounds of arrays, and storage class attributes. Files must be described in terms of the constituent records, the logical and physical organization, and the appropriate method of access. The requirement for formatted input/output predicates the specification of the external form in which the data exists (for input) and/or is to be edited (for output).

Problem control data, such as label and event data, must be declared. In addition, all subprogram units (procedures, subroutines, and functions) must be declared in terms of the statements to be executed at the time of invocation, the arguments, associated attributes, and entry points.

AED — Data Declarations. Data declarations are ALGOL-like, in the sense that grouping is generally by property--e.g., property variable, variable, . . . $. Thus, a variable may appear in several such lists. Properties include data types, packing instructions for partial-word data, the fact of being a procedure, label, bead component (with offset from base), array (with associated bounds), EXTERNAL/COMMON, and PRESET instructions (initialization code to be executed interpretively by the compiler).

File Declarations. File definitions are not present in AED. The language, as such, does not contain any concept of I/O--in the best ALGOL tradition. File characteristics are described by settings of items in file-associated beads, which are passed to I/O subroutine packages.

Format Descriptions. AED has no format descriptions.

Procedure/Subroutine/Function Definitions. In AED, a function is a typed procedure. All procedures are defined by the construct:

DEFINE <procname> (<arglist>) WHERE <argdeclarations> TOBE--).

Within a procedure, procedures to be called are declared as data items (PROCEDURE GLITCH, KLUDGE, SQRT$,).

Such definitions and declarations include the procedure type, if any, and RECURSIVE, if applicable, but declarations in the calling procedure do not mention the number or type of parameters; arguments, when passed, are assumed to match properly (see C.1.b.7).

COBOL

Data Declarations. All data except unnamed constants must be described in the DATA Division in either the FILE, WORKING-STORAGE, CONSTANT, LINKAGE, or REPORT Section. Data entry items are specified by a level number, data name or FILLER, PICTURE clause, and optional JUSTIFIED, OCCURS, SYNCHRONIZED, USAGE, REDEFINES, and VALUE clauses. The assignment of level numbers is as follows:

01-49	regular levels
66	for RENAMES clause
77	for noncontiguous data items

88 for condition names (follows the conditional variable)

FILLER is a reserved word used to specify a data field to be unnamed in record descriptions. The descriptive clauses are used to specify repetition, data justification (position changes), data form in the machine, and the redefining/redescribing of data entry items. Redefining data may not have VALUE clauses and must not exceed the total length of the data entry being redefined. Data editing is indicated by the description given and all operations on the data are governed by the description clauses.

File Declarations. File descriptions are first stated in the ENVIRONMENT Division, where the internal file name is associated with the external device name or code number and the attributes of the hardware device. The full description of the file in the DATA Division begins with a description indicator: FD for file description, SD for a sort file description (sort files are considered as internal data files), and RD for a report file description. The information that may be specified with this descriptor are the file name, labeling specs, record length, block size, recording mode, data record names, and character count of the record. Record structure format is used to describe the data format for the file, where level number 01 indicates the start of a new record description. All files must be identified in the ENVIRONMENT Division by using a SELECT clause with optional OPTIONAL, RESERVE, FILE-LIMIT, ACCESS MODE, PROCESSING MODE, and ACTUAL KEY clauses. These clauses respectively indicate that the input file (accessed sequentially) may not necessarily be used; specify the number of buffer areas to be reserved; specify the logical

beginning and end of a file on a mass storage device; specify the accessing and processing modes; and provide the key for records processed on a randomly-accessed file. Other options specified in the I-O CONTROL Section are: RERUN which produces a checkpoint record; SAME AREA which indicates two or more files use the same core storage area; and MULTIPLE FILE TAPE which indicates that more than one file is on a reel of tape.

Format Descriptions. Format descriptions are specified in the DATA Division in any or all of the allowable sections. Input formats specified in the FILE Section or the WORKING-STORAGE Section are described by use of level numbers, data names, fillers, and picture clauses which specify data by 9, A or Xs--the filler pictures being restricted to X. The record structure is used. Output formats may be specified as are the input formats, with an additional capability provided in the REPORT Section. The RD entry of this section describes the physical report format, and the record group entries describe the data items. Specifications include the maximum number of lines per page, the position of the report group on a page, and the data items to be used for control of HEADING, FOOTING, FIRST DETAIL, and LAST DETAIL. The programmer may also indicate the number of lines to be skipped before output formatting is to continue. The REPORT Section provides output formatting as the FILE Section provides for input and output formatting in a record structure, but with the additional format control of output report positioning controls.

Procedure/Subroutine/Function Definitions. COBOL does not require an explicit declaration of a procedure, subroutine,

or function definition. The only functions considered to be declared are those specified in the ENVIRONMENT Division as SPECIAL-NAMES. These serve to interact with the system and return information to the program, such as time of day, date, and input/output status. There is one form of a PROCEDURE Division statement that indicates a COBOL program is a subprogram; this is the USING clause. The USING clause, which may appear in a CALL, ENTRY, or PROCEDURE Division statement, makes data defined in the calling program available to the called program.

JOVIAL

Data Declarations. Data definitions are initiated by one of the key words ITEM, STRING, ARRAY, and TABLE, which is followed by a name and a list of attributes.

ITEM and STRING data may be declared floating, integer, fixed, dual (F, I, A, D), or Hollerith, Transmission (H, T), or status (S), or Boolean (B). Sizes may be specified in bits or bytes, as appropriate. Specifications for numeric data may include ranges, rounding, and signed/unsigned formats. Specifications for each status data include the status constants which are legal for the data.

ARRAY data declarations include the number and size of each subscript and the data type (F, I, A, D, H, T, S, or B); ARRAY-defined arrays are limited to a single data type. Elements of the array may be initialized with a BEGIN...END constant list.

TABLE data declarations may include the type (rigid or variable), number of entries in the table, the organization

(parallel or serial), the packing (none, medium, or dense), and the elements of the structure with optional lists of initializing constants. The element list is bracketed by BEGIN and END.

Items may be defined for local use by simple reference to a name which has no corresponding declaration; such items are integer type, unless otherwise specified with MODE. This is the only provision for default attributes of data.

Tables can be declared 'like' already-declared tables In general, this provides for duplicating structures without having to write out the structure's elements. Different size, organization, and packing specifications may be provided; if not, the original specifications are used. The name of the like table is the original table name with a single letter or numeral suffix; all data names of the like structure are then similarly suffixed.

Tables can be overlayed to occupy specific storage addresses, to occupy the same storage as another table, or to occupy storage adjacent to another table.

Data (except simple items) must be declared prior to their first use in a program. Once declared, the name is known to all levels of the containing procedure.

File Declarations. File descriptions begin with FILE, followed by the filename and the file attributes. These include data type (Hollerith or binary), the estimated maximum number of records in the file, rigid or variable record size,

the estimated maximum number of bytes (Hollerith) or bits (binary) in a record, a list of all possible file states as status constants, and the device name.

Format Descriptions. Format descriptions are not provided.

Procedure/Subroutine/Function/Definitions. Procedure-type (functions, subroutines, procedures, etc.) definitions are introduced by one of three key words: 'PROGRAM, CLOSE, and PROC.

'PROGRAM, the only one of the three for which code does not form part of the definition, is used to identify external procedures. This definition takes the form:

'PROGRAM <procname> [<numericorigin>] $

where <procname> is the procedure identifier and <numericorigin> is an optional decimal or octal absolute core location (a feature which does not lend itself to relocatable environments). <procname> is invoked by statements of the form:

GOTO <procname> $

with control normally returning automatically to the statement following the GOTO.

CLOSE procedures are similar to 'PROGRAM procedures except that CLOSE procedures include executable code as part of the definition, which takes the form:

```
CLOSE <procname> $
       BEGIN
      <statementlist>
       END
```

where <procname> is the procedure identifier and <statement-

list> is a series of statements. <procname> is invoked by statements of the form:

GOTO <procname> $

with control normally returning automatically to the statement following the GOTO.

PROC procedures are the most complex of the procedure definitions and are used both for functions and for procedures. They may include data declarations as well as executable code in the form:

PROC <procname> [({ <inputparameterlist> / =<outputparameterlist> / <inputparameterlist> = outputparameterlist })] $

[<declarationlist>]

BEGIN

<statementlist>

END

The <inputparameterlist> consists of a series of simple item names, names of arrays, names of tables, or names of CLOSE procedures (CLOSE names are followed by one or more spaces and a period). The <outputparameterlist> consists of a series of names of simple items, names of tables, names of arrays, or statement labels. Data used in the procedure may be defined three ways: globally, by its definition in a procedure containing the using procedure; locally, by its definition in the using procedure; and by default (for simple items only) by its use in the procedure without an explicit definition, either local or global. The <statementlist> must not itself contain procedure definitions. A procedure is identified as a function-type procedure by the absence of an <outputparameterlist> and by the presence, in the <declarationlist>, of a simple item declaration with the

same name as the <procname>. Procedures are invoked by statements of the form:

<procname> [[({ <inputparameterlist> | = <outputparameterlist> | <inputparameterlist>=<outputparameterlist> })]] $

Functions are invoked by the appearance of the <functionname> as part of an expression in the following form:

. . . <functionname> ([<inputparameterlist>]). . .

Note that the parameter parentheses are optional in the procedure invocation, but mandatory for function invocations.

There are two built-in procedures whose definition is implied: REM, a remainder function, and REMQUO, a remainder and quotient procedure. These are not reserved words and may be redefined for other uses.

Other built-in procedure-like features include ABS, ALL, BIT, BYTE, CHAR, ENT, ENTRY, 'LOC, MANT, MENT, NWDSEN, OFF, and POS. All these names are reserved words; the operation of these features is discussed in detail in other sections (see A.4 and C.1.b.6).

PL/I

Data Declarations. All explicit declarations in PL/I begin with the key word DECLARE, followed by the name of the item and the attributes of the item. For data declarations, the attributes include type descriptors and precision descriptors. Some of the key word type descriptors are: FIXED, FLOAT, BINARY, DECIMAL, CHARACTER, POINTER, LABEL, etc.

For arrays, the upper and lower bound (lower bound is 1, if not included) follow the array name directly (e.g., DECLARE A (-2:4,3) BINARY FLOAT (12);) with the two numbers

connected by a colon. As many as 32 dimensions may be declared for an array. Arrays default to the same characteristics as simple variables of the same name.

File Declarations. For file declarations, the filename is followed by the key word FILE and the attributes of the file. These attributes include: STREAM, RECORD, PRINT, SEQUENTIAL, DIRECT, KEYED, UNBUFFERED, etc. The default attributes for files are established when the file is first used and depend upon the use made of the file (i.e., a READ statement causes the file to default to an INPUT file).

Format Descriptions. PL/I provides both local and remote format description statements for use with edit-directed I/O. Format items may either be remote or part of the edit-directed I/O statements (e.g., PUT EDIT ('A')(A(1));). For non-edit directed I/O, format items are not used in PL/I. Declarations may appear anywhere in the program, but labeled format statements must be in the same block with all its references.

Procedure/Subroutine/Function Definitions. External subroutines and functions are declared by following the function name with the key word ENTRY, followed by the data types of parameters passed, if any. Subroutines and functions begin with a PROCEDURE statement and are completed by an END statement. Control passes back to the calling program whenever the final END statement or a RETURN statement is encountered.

C.1.a.3 Program Structure - Non-executable Units - Storage Allocation/Segmentation

Storage allocation declaratives provide the capability to reserve storage for static variables and also, in more advanced languages, to specify the span of program execution over which dynamic variables are to be in existence.

In addition, the efficent use of restricted memory is greatly enhanced if the language provides the capability for program segmentation (i.e., the partitioning of a program into chronologically independent modules which can be loaded into main memory when and as needed). Storage allocation declaratives in the source program provide the means whereby information necessary for the segmentation/overlaying process is supplied to the compiler.

For the multiprogrammed environment of most present-day systems, the ability to control storage allocation dynamically permits more jobs to be run simultaneously with a corresponding increase in system throughput, but at increased costs in system overhead. Dynamic control of storage, however, is critical for some features of modern systems, notably recursion and data structure building.

AED — Except for dynamically-allocated beads (see C.1.b.2.e), variables declared in an AED program are normally allocated at load time; i.e., FORTRAN-like or STATIC in PL/I. In a recursive procedure, local variables declared OWN are stacked with each invocation; i.e., AUTOMATIC in PL/I. Note that OWN thus has precisely opposite meanings in AED and ALGOL.

COBOL — COBOL has no real dynamic storage allocation except for the OCCURS DEPENDING ON clause which may be used for data specification; in which case, the actual storage used depends on another defined data item value. The SELECT OPTIONAL file option will also change the allocation of storage at execution,

depending on the presence of the selected file. Another means of controlling storage allocation is by using the SAME AREA clause for files; only one file may be open at a time, but all files specified with this clause will occupy the same storage locations. Data entries described with REDEFINES provide the user with a technique for describing the same data area with different attributes. Procedure coding will be governed according to the data name used to reference the storage cells.

COBOL provides an internal method for controlling segmentation by communication with the computer. Segmentation deals only with procedures. If this option is used, the entire PROCEDURE Division must be written in sections, with each section classified as fixed or independent segments of the program. A fixed section is composed of permanent segments and overlayable segments. Independent sections can overlay and be overlayed. Priority numbers, 00-99, are assigned with the section name; 00-49 are for the fixed segments and 50-99 are for the independent segments. Segmentation limitations are established in the ENVIRONMENT Division.

JOVIAL

Data storage allocation is done at compile time with statements of the form:

OVERLAY { <datalist> | <numericorigin>=<datalist> }$

where <datalist> is a list of simple item names, table names, or array names; members of the list are separated by an equals sign to indicate common use of storage and by a comma to indicate adjacent use of storage. <numericorogin> is a decimal or octal absolute core location, a feature which does not lend itself to relocatable environments.

Program storage allocation is only available for external procedures (see C.1.a.2) with statements of the form:

'PROGRAM <procname> [<numericorigin>] $

PL/I

Storage is allocated for all static variables before the program begins execution. For automatic variables, storage is allocated upon entering the block in which the variable is declared, except for those variables declared to be either BASED or CONTROLLED. For these variables, the user must allocate storage within the program by use of the ALLOCATE statement. Therefore, the user has the capability of creating push-down stacks, allocating arrays whose size varies with each run, or creating various trees and rings. User-allocated storage may be released via the FREE statement.

C.1.a.4 Program Structure - Non-executable Units - Environment/Operating System Descriptions

A program may be conceptualized as executing in an environment constituted of the hardware components which make up the total computer system, and under the control and management of the operating system. The environment of a computer program consists of both hardware and software elements. Hardware elements may include instruction sets, memory storage, I/O channels, I/O devices, and data codes. Software elements may include task initiation and monitoring facilities, resource management, data management, sort/merge capabilities, and procedures for recovery from hardware or software failures. Consequently, more sophisticated programming can be accomplished when languages provide for declarations and specifications whereby the hardware may be described and/or interfaces with the operating system established.

In varying degrees, features exist in languages which allow for the specification of the hardware configuration (both compilation and

execution), the specification of corrective action to be taken when hardware interrupts occur, and the ability to obtain status information when operating in a multiprogramming/multiprocessing environment. The ability to define such aspects of a program's environment can have significant effect on the efficiency of object code produced by a compiler.

AED

AED does not provide any facilities for describing either the environment or the interaction with the operating system.

COBOL

Environment and operating system descriptions are specified in the second division; i.e., ENVIRONMENT Division. This division is coded to specify the necessary external-internal relationships. In this division, the computer is specified both compiling and executing; the input units are named and described, as well as access methods to be used.

The reserved sections associated with this division are CONFIGURATION and INPUT-OUTPUT. In the CONFIGURATION Section the SOURCE-COMPUTER, OBJECT-COMPUTER, MEMORY SIZE SEGMENT-LIMIT, SPECIAL NAMES, and COPY options are specified. Hardware specifications are also described with any implementation names. The INPUT-OUTPUT Section allows for the specification of file names, file COPY, file renaming, buffer allocations, optional file presence, tape file positioning, random access file limits, processing mode, and multiple files.

JOVIAL — JOVIAL makes no provision for explicit descriptions of operating system interfaces.

PL/I — Environment descriptions in PL/I are found in the environment attribute of a file declaration and the options parameter of a procedure statement. The options of the environment attribute of a file declaration include record format, buffer allocation, data set organization, etc. The options parameter must be MAIN for a main procedure on an IBM machine.

C.1.b.1 Program Structure - Executable Units - Expressions

Section A of this appendix presents the basic atomic elements of the subject languages; the manner in which these elements may be combined to form larger executable units, which range from expressions through comprehensive procedures, will be the subject of Section C.1.b.

An expression constitutes the lowest level executable aggregate and may be defined, in general terms, as a series of operands connected by operators in accordance with a language-specific set of rules. Expressions may have subexpressions nested to any arbitrary depth. Conventions are established to specify the precedence of operators and the order of evaluation of parenthetical subexpressions. Languages differ in the extent to which expressions may involve variables of mixed mode (e.g., floating-point, integer, and logical) and the complexity of the constructs allowed for subexpressions.

AED

Basically, conventional arithmetic and Boolean expressions, with a defined operator precedence and parenthetical nesting, are permitted. Beyond this, the structure of AED is largely built upon the concept of 'type transformation' and 'phrase substitution'. What this means is that a phrase (expression) whose type is, say, 'integer' by the type transformation rules may be substituted wherever an integer may appear. Typed values are defined not only for the usual kinds of phrase (e.g., function references and expressions) but also for constructs not usually thought of as having a value (e.g., whole assignment statements, IF-THEN-ELSE constructions, BEGIN-END groups). Thus AED is characterized as an 'expression language'; a whole program is simply the pinnacle in a hierarchy of expressions; for example:

```
X = A = IF P AND Q THEN B
ELSE INTEGER BEGIN C = D $, E = F END
```

which will set X and A equal to F (the value of the BEGIN-END group) if either P or Q is FALSE, and to B otherwise.

This approach leaves very little that cannot be written and have some valid interpretation. In particular, mixed mode arithmetic is allowed (an INTEGER may be written wherever a REAL may be in an expression) and subscripts and arguments may be arbitrarily complex--whole programs, even.

COBOL

Expression rules all have one thing in common; that is, all data items referenced must be uniquely identified by a unique name or a qualified name. No name may be both a data name and a procedure name. A data name used as a qualifier for another data name may not be subscripted.

Subscripts may be a numeric literal, an integer, TALLY, or an identifier (unique data name). The value of a subscript must be a positive integer and the subscript itself enclosed in parenthesis. The format is:

[<data-name-1>] [IN <data-name-2>] [IN <data-name-3>] (<subscript>)

Index names are initialized by a SET verb and may be used for direct indexing, when the index name serves as a subscript, or may be used for relative indexing, where the index name is followed by a plus or minus and an integer numeric literal. Index names must be enclosed in parentheses when being used for table referencing. The format is similar to that shown for subscripting.

Arithmetic expressions may be composed of identifiers of numeric elementary items, numeric literals, arithmetic operators, and arithmetic verbs. Parentheses may be used for clarification and to specify the hierarchy of operation. Any arithmetic expression may be preceded by a unary operator. The language allows for combining arithmetic

operators, receiving data fields, and operands, without restrictions. If parentheses are not used, evaluation proceeds from left to right for operators of equal value, where operator hierarchy is unary plus/minus, exponentiation, multiply/divide, and add/subtract.

When parentheses are used, the evaluation proceeds from the least inclusive enclosed set to the most inclusive set, applying the same operator hierarchy rules.

Conditionals have two mode rules: (1) Relational conditionals may have mixed mode operands regardless of declared USAGE; (2) All other conditionals must have the same declared USAGE. For numeric data, the test is made on the algebraic value of the operands, with zero being unique regardless of the presence of a sign. Nonnumeric operands are processed according to an implementation-defined collating sequence.

Combined conditionals allow, except for the first operand and operator, for the omission of the subject and the relational operator of the relational condition. This causes the omitted subject to be replaced by the last preceding subject stated or omitted; the omitted operator is replaced by the last preceding stated relative operator. There is some possible ambiguity with NOT; the expression a>b AND NOT>c OR d is evaluated as a>b AND (NOT a>c) OR a>d.

The hierarchy for conditional expressions is arithmetic expressions, all relational operators, NOT, AND, OR.

JOVIAL

Expressions (known as formulas in JOVIAL) may be one of six types: literal, status, entry, numeric, dual, or boolean. Literal formulas are limited to an octal constant, or a constant, variable or function which is either Hollerith or transmission code. Status formulas are limited to a status constant, variable, or function. Entry formulas are limited to the value 0 or an entry variable. Numeric and dual formulas may consist of numeric and dual (respectively) constants, variables, or functions, and these may be combined in standard ways with arithmetic operations and parentheses. Both types of formulas may involve + or - prefixes, as well as absolute and exponentiation operations. Only numeric formulas may include the NWDSEN, NENT, and 'LOC functions. Boolean formulas consist of formulas of the other five types bound to each other by relational and logical operations, as well as boolean constants, variables, and functions.

Evaluation order applies only to numeric, dual, and boolean formula types. In all these formulas, values of constants, variables, or functions are determined left to right before any other evaluation is done, with parameters evaluated before their functions, and indices before their indexed items. Parentheses then have the standard grouping effect, with inner formulas evaluated before outer ones. Arithmetic operators have the following priority: first negation, then exponentiation, then multiplication and division, last addition and subtraction. Boolean operators have a lower priority than arithmetic operators.

Evaluation of numeric formulas can involve mixes of data types (integer, fixed, floating, or octal). In

general, arithmetic operations involving at least one floating value are done in floating form.

Evaluation of boolean formulas is carried out only as far as necessary to establish the truth of the result.

In general, relational operators can relate operands of the same type, either numeric, dual, literal (only EQ or NQ for hollerith), status or entry (only EQ or NQ). As a special case, file names may be related to status formulas.

PL/I

In PL/I, an expression is either an element expression, an array expression, or a structure expression, depending on whether it represents an element, array, or structure value. Array variables and structure variables cannot appear in the same expression, but element variables and constants may appear in any expression.

Single constants or variable expressions may appear anywhere in a program, and PL/I allows expressions anywhere, if their evaluation yields a valid value.

There are four classes of operations acceptable for use in expressions: arithmetic, bit-string, comparison, and concatenation. An arithmetic operation combines operands with one of the following operators: +, -, *, /, **. The result of the operation is dependent on the operands. A bit-string operation employs one of the following operators, ¬, & and | . The operands are converted to bit strings and the shorter string is padded on the right with zeroes, if necessary. The result is equal in length to the longest

operand. The comparison operators include: <, ¬<, <=, =, ¬=, =>, >, ¬>. There are three types of comparisons, depending on the operands: algebraic, character, and bit. The concatenation operation combines operands with the concatenation symbol ||. The result is a string whose length is the sum of the length of the two operands.

C.1.b.2.a Program Structure - Executable Units - Statements - Data Manipulation

A programming statement constitutes the most elementary complete specification of an action or process to be executed by the computer. In general, statements take the form of imperatives for data manipulation, of decision-making conditional tests to be performed, and of commands for program sequence control. Statements may be labeled, and thus serve as an identifiable program location to which control may be transferred.

Data manipulation statements may be categorized as:

1. assignment
2. data movement
3. list processing
4. string editing
5. sort/merge

Assignment statements are used for formula evaluation and assign the resultant value(s) to a named variable. Some programming languages limit such operations to simple scalar quantities, while others allow operations on more complex data aggregates such as matrices, structures, and files.

Data movement statements, as the name implies, constitute that class of statements which allow for the mass transfer of data from

one data structure to another; a convenience, per se, but a necessity for those programming languages designed for data processing applications involving large quantities of data.

List processing statements are basically operations with pointers and offsets, and provide the mechanisms to access and manipulate those data items whose transitory characteristics make inefficient normal methods of storage deriving from the linear nature of computer memory.

String editing statements encompass that class of statements which provide for operations on alphanumeric data. Representative of such operations are concatenation, decomposition, and pattern searching.

Sort/merge statements, such as those that exist in COBOL, represent a special form of data manipulation, that is more conventionally accomplished with utility packages.

AED

Assignment Statements. AED is an 'expression language,' which means that all complete expressions--function references, arithmetic expressions, whole statements, IF-THEN-ELSE constructions, etc.--have a value (see C.1.b.1). At any point, such a value may be assigned as soon as computed, by substituting for the expression <exp> the expression <var> = <exp>. Thus, one can have conventional assignment statements and much more complex forms as well.

Data Movement Statements. At the explicit language level, internal data movement is strictly by assignment of elementary data items. Movement of aggregates may be effected by function call--e.g., the standard procedures

SPRAY and GLUE are used to move and transform the format (packed or unpacked) of character strings.

List Processing Statements. List processing, at the language level, is a 'do-it-yourself' proposition using the elementary pointer operations. However, a standard data structure package, a prime example of the AED system-building philosophy, permits defining very general mechanisms for processing of lists or other structures.

String Editing Statements. As noted previously (B.1.a.3), AED has no string handling facilities at the language level. A standard function package does exist, however, for moving, substring extracting and replacing, scanning, comparing (etc.) strings 'conceptually' implemented underneath the POINTER and INTEGER types. Unfortunately for clarity or conciseness, this package requires rather too much explicit attention by the programmer to the mechanization itself.

Sort/Merge Statements. AED, as such, has no language facilities or major language-related packages for sorting or merging.

COBOL

Assignment Statements. Assignments in COBOL may result from using the equal symbol or from the storage of the results of arithmetic verbs (ADD, SUBTRACT, MULTIPLY, DIVIDE), with the GIVING option or the cumulative TO option; for example: ADD 1 TO B or MULTIPLY A BY B GIVING C. There are other options which will allow the results to be ROUNDED and/or tested for SIZE ERROR. The COMPUTE verb also serves as a method of assignment. This verb allows for computation of more complex arithmetic statements.

These arithmetic assignment statements may be used with a CORR/CORRESPONDING option, which provides a method of assignment and use based on identical data names in different structures.

Data Movement Statements. Data movement is accomplished by the MOVE verb and the MOVE CORR/CORRESPONDING verb. Movement may be at the elementary level or at any point between and including the elementary and record (01) level. Special data movement is accomplished with the INITIATE and GENERATE statements used for report writing. These statements cause data items described in report groups to be moved, without specifying the entire data list by name. In the DECLARATIVES Section of the PROCEDURE Division, some data movement may take place (refer to I/O descriptions).

List Processing Statements. Lists of data are processed by the PERFORM verb and the special table-handling verb, SEARCH.

String Editing Statements. The EXAMINE verb provides a means of direct string editing, while REDEFINES provides a means of string redefining for subsequent manipulation. The examination process allows for character tallying and replacing with options of ALL, UNTIL FIRST, LEADING, and FIRST. Other editing of string data may be accomplished by moving one string to another structure, described with different edit control characters in the picture clause.

Sort/Merge Statements. COBOL provides a SORT feature which allows for sorting and handling of data before and/or after

the sort. The operation requires specifications in the ENVIRONMENT Division for the input and output files, in the DATA Division for the sort-file description (SD), and in the PROCEDURE Division for the SORT statement and the ascending or descending sequence. A program may have more than one SORT statement. Special care should be given to the collating sequence of sort keys, since this sequence is not a COBOL specification. Sort keys may not have an OCCURS clause nor be subordinate to an OCCURS clause entry; keys must be at a fixed displacement from the beginning of the record. The two procedures, INPUT Procedure and OUTPUT Procedure, associated with SORT may not contain any SORT statements and may not transfer control outside of the procedure (USE declaratives are not considered transfers of control); the other parts of the main program must not transfer control to points inside of these procedures. The INPUT Procedure must have at least one RELEASE statement to release the sorted record. The USING and GIVING options provide for transferring of data files to and from the sort files for sorting.

JOVIAL

Assignment Statements. Assignment statements have the form:

<variable> = <expression> $

where <variable> and <expression> must both be numeric, dual, literal, boolean, status, or table entry. JOVIAL also has a special form of assignment statement, the exchange statement, with the form:

<variable> == <variable> $

where both <variable>s must be one of the types listed above.

Other Data Manipulation Statements. JOVIAL has no statements for data movement, list processing, string editing, or sort/merge.

PL/I

Assignment Statements. The assignment statement assigns the value of the expression on the right side of an equal sign to the variable on the left of the equal sign (e.g., A=B; assigns the value of B to the variable A). More than one variable may be assigned the same value at once, but each variable must be separated by commas (e.g., A, B, C = D;). Entire arrays may be assigned values in one statement (e.g., DCL A(5), B(5) BIN FIXED; A = B;) or one 'slice' of an array can be assigned by using a * in that parameter (e.g., DCL A(5,2), B(5) BIN FIXED; A(*,1)=B;). Structures can be assigned values from other structures or they can be assigned with the option, BY NAME, which transfers data from one structure to the other wherever the names of lower levels agree.

Data Movement Statements. There are no other data movement statements in PL/I.

List Processing Statements. There are no explicit list processing operations in PL/I, but the user can create his own by use of pointer data manipulation.

String Editing Statements. There are two string-editing operations in PL/I. The first is the concatenation operation, which concatenates two strings together (i.e., '0'||'1101' yields '01101'). The second is the SUBSTR function, which yields a continuous substring of the original

string. The first parameter of SUBSTR is the original string, the second is the number of the first character of the resultant string, and the third parameter is the length of the resultant string (e.g., SUBSTR ('ABCD',2,2) yields 'BC').

C.1.b.2.b Program Structure - Executable Units - Statements - Program Sequence Control

A natural dichotomy exists in programming languages between those statements that actually accomplish the processing of data and those that maintain the proper logical sequence of program execution.

Branching instructions of the form 'GO TO <label>' provide the simplest means of effecting an unconditional transfer of control and thereby provide the most common mechanism by which normal sequential execution of programs may be altered.

Additionally, most programming language instruction sets include conditional statements which provide the capability to select certain parts of the program to be executed to the exclusion of others, based on the outcome of specified execution-time tests. Languages differ in the degree of complexity allowed in these tests, but in the most generalized form alternative paths are specified for execution, according as the stated condition evaluates to true or false; for example, 'IF <condition> THEN <label-1> ELSE <label-2>' is representative.

The ability to execute iteratively a sequence of computer instructions is a prime requisite for programming languages, due to the repetitive nature of data processing operations. Such a sequence of instructions is defined as a loop, and is executed repeatedly

under the control of a loop control statement with address modification changing the operands for each iteration. Loop control statements specify the range of the loop and also the parameters for the initialization, modification, and testing of the loop index. The number of times the loop body is to be executed may be stated in various forms: explicitly, as a range of values which the loop index is to assume, or implicitly, until a terminating condition is satisifed.

A final class of program sequence control statements are statements such as STOP, END, and HALT which provide overall program execution control.

AED

Unconditional Branches. AED has a 'GOTO <label-or-switch>' and also the standard function 'DOIT (<pointer>,...)', where <pointer> may be to a label as well as procedure. In either case, branching may be to any point in the current block or in any currently active block.

Conditional Branches. Conditional branches are handled by making the GOTO or DOIT one of the consequent clauses of an IF-THEN-ELSE compound statement.

Iteration Control. Iteration is controlled by FOR...STEP... UNTIL...DO, FOR...STEP...WHILE...DO, or FOR...WHILE...DO clauses. The STEP [...UNTIL]construction controls incrementing an integer variable up to some maximum. WHILE takes a Boolean expression; when the expression evaluates as FALSE, the iteration terminates. Testing for the termination condition is done at the beginning of each iteration; i.e., it is natural to iterate zero times.

Execution Control. Procedure call is by mention of the procedure name followed by a parenthesized (though possibly empty) argument list. Return is by encountering the end of a procedure or GOTO RETURN. RETURN acts as a reserved label. It is also possible to return through several call levels at once.

COBOL

Unconditional Branches. Program sequence control in COBOL is controlled by key words and optional key word clauses. Unconditional branching is accomplished by the GO TO verb followed by the paragraph label.

Conditional Branches. Conditional branching occurs as the result of computational verbs with the SIZE ERROR clause, a GO TO verb with the DEPENDING ON some specified condition option, and the IF statement. The following verbs have both a conditional and an unconditional mode depending on the associated options: the READ with END option; the RETURN with END option; the SEARCH verb associated with table handling methods; the CALL external procedures verb; the ALTER paragraph labels verb; the PERFORM internal sections or paragraphs verb; and the EXIT statement. The DEPENDING ON clauses allow the value of a single identifier to determine to which procedure/paragraph control will be transferred. The ALTER statements allow the destination of a GO TO to be changed; the paragraph with the destinator being changed must contain only one sentence, the GOTO sentence. Statements altered must not be in segments categorized as independent (overlayable or overlays).

Iteration Control. Iteration is accomplished by the PERFORM verb with the options of executing a specified series of procedures/paragraphs; executing a series of procedures a specified number of TIMES; executing UNTIL a condition is satisfied; executing a series of procedures VARYING a variable or index name FROM a starting value BY an increment value UNTIL a condition is met (three such levels of iteration are allowed).

Execution Control. Execution control of the COBOL language is comparable to the English language in that all sentences are executed from the start of the PROCEDURE Division to the end in a sequential manner, unless unconditional branching occurs. Even if a paragraph serves as a procedure, unless explicit transfer of control takes place, this paragraph will be executed as it is entered.

Generally sentences are evaluated from left to right.

JOVIAL

Unconditional Branches. In general, sequence control in a program is left to right, top to bottom. Conditional and unconditional branches and loops provide for exceptions to this order.

Unconditional branches in JOVIAL are handled by statements of the form:

GOTO <label> $

where <label> may be a statement label or the name of a CLOSE or 'PROGRAM procedure. If transfer is to a procedure, normal termination of the procedure (by RETURN) will cause control to resume with the statement following the GOTO.

Conditional Branches. Conditional branch statements fall into one of two categories: conditional and alternative statements, and switch statements.

Conditional statements have the form:

IF<relationalexpression>$

where <relationalexpression> involves expressions, relational operators, and logical operators. If the <relationalexpression> is true, the next statement is executed; otherwise it is skipped. Evaluation of the <relationalexpression> stops as soon as its truth or falseness can be established.

Alternative statements have the form:

```
IFEITH<relationalexpression>$
 <statement>
[  ORIF<relationalexpression>$      ]
[  <statement>                      ]
[ [...ORIF<relationalexpression>$ ] ]
[ [  <statement>                  ] ]
[ END                               ]
```

where <relationalexpression> is as described above and each is tested until one is found true, whereupon the associated statement is executed, or until all are found false, whereupon control passes to whatever follows the END. Testing may start within an alternative statement by control being passed directly to a labeled ORIF.

Switch statements involve GOTOs and either item switches or index switches.

Item switches are set up by:

SWITCH<switchname>({ <filename> / <itemname> }) = (<value1>=<label1> [,[<value2>= <label2>]]...)$

and used by:

GOTO <switchname>$

where, if a specified <itemvalue> or <filename> status value are not available, control passes to the next statement in order. SWITCH statements should be located where control cannot reach them by normal sequencing.

Index switches are set up by:

SWITCH<switchname> (<label1>[,[<label2>]...])$

and used by:

GOTO <switchname>($ <expression> $) $

where, if the <expression> value does not have a corresponding label (either because of adjacent commas or too short a label list), control passes to the next statement in order.

Iteration Control. Iteration is provided by FOR statements of the form:

FOR<loopvariable> = { ALL (<name>) / <expression1>[,<expression2>[,<expression3>]] }$
<statement>

where <loopvariable>s are single-letter names (e.g., I), <expression1> gives an initial value, <expression2> an increment value, and <expression3> a terminal value. Incrementing and testing is done at the end of the <statement>'s execution (the <statement> can be a BEGIN-END compound statement). Manipulation of the <loopvariable> can be done by assignment statements within the <statement>. Control can be taken out of the loop to the increment

and test portion at any time with a statement of the form:

TEST [<loopvariable>] $

Execution Control. Subprogram linkage is provided by:

RETURN $

for return to calling procedures, and by:

STOP $

for return to the system. There is no provision in JOVIAL for event-based linkage.

PL/I

Unconditional Branches. The GO TO statement unconditionally transfers the flow of the program to the label designated (e.g., GO TO LABEL1;).

Conditional Branches. The IF statement tests the value of an expression and controls the flow of execution according to the result. The expression is converted to a bit string and if any bit in the string is 1, then the THEN unit is executed, otherwise the ELSE unit is executed (e.g., IF <expression> THEN <unit1>; ELSE <unit2>;), where a unit is defined as a single statement or a statement group. If no ELSE unit is specified and the bit string is all zeros, control passes to the next statement.

Iteration Control. There are two types of iteration statements. The first is of the form:

```
DO     WHILE (<expression>);
<unit>
END;
```

All statements between the DO and the END are executed repetitively as long as the expression reduces to a bit string where at least one bit is 1.

The second form of an iterative statement is:

DO <variable> = <expression-1> TO <expression-2> BY <expression-3>;
<unit>
END;

In this case, the <variable> is assigned <expression-1>. When control reaches the END statement, <expression-3> is added to the <variable>. If [[<expression-3> ≥ 0] and [<variable> > <expression-2>]] or [[<expression-3> < 0] and [<variable> < <expression-2>]], then transfer to the statement after the END statement, otherwise continue. If <expression-3> is missing, it is assumed to be 1.

The iteration statements can be combined as follows:
DO <V> = <e1> TO <e2> BY <e3> WHILE (<e4>);

Execution Control. There are a number of statements in PL/I that could be characterized as execution control statements. These include STOP, EXIT, RETURN, WAIT, CALL, and ON statements. The STOP statement causes immediate termination of all tasks. The EXIT statement causes termination of the current task and all subtasks. The RETURN statement returns control to the calling procedure and it may also return a value, if the called procedure is a function (e.g., RETURN (<expression>) ;). A block may retain control until certain events are met by use of a WAIT statement. CALL statements cause transfer of control to a specified entry point of a procedure. Finally, ON statements specify action to be taken when a specific interrupt occurs.

C.1.b.2.c Program Structure - Executable Units - Statements - Input/Output

Input-output statements provide the facilities whereby the transfer of data between an external storage medium and the internal storage of the computer may be accomplished. These I/O statements are the communication mechanism between the computer proper and the 'outside world', and effect the translation of the data into the form required by the direction of the data transfer (i.e., internal→ external, or vice versa). The essential components of an input/ output statement consist of a command (e.g., READ, WRITE, GET, PUT) and a list of variables for transfer. Some languages require the specification of the physical media (e.g., tape, cards, disk), but program transferability is enhanced to the extent that the I/O statements are device-independent. Most programming languages provide for two modes of input/output, namely, formatted and unformatted. Unformatted I/O operations generate no editing; the external and internal representation of the data elements are essentially the same. In contrast, formatted I/O allows the user to describe, via format specifications, the form of the data as it exists on the external media (input), or as it is to be edited for external presentation (output), where the external media is frequently card images or printed reports.

Input/output operations have also been categorized as stream-oriented and record-oriented. Stream-oriented input/output is based on the concept of data existing as a continuous stream of characters. Each data item undergoes the appropriate conversion during processing of the stream. Stream-oriented transmission is required when the input to be processed has been prepared on an external device (e.g., symbolic punched cards) and, conversely, stream-oriented transmission must be specified if the output data is intended for report generation

and human interpretation.

Record-oriented transmission is based on the concept of data existing in aggregates, called records. Transmission is accomplished on a record-by-record basis, with no data conversion operations. The internal and the external representation of the data are exact duplicates. Record-oriented transmission allows for the rapid transfer of data between the hardware elements of a computer system and for the communication between programs.

Stream-oriented I/O has the advantage of machine-independence; while record-oriented I/O, which is machine-dependent, derives its advantage from the efficiency and speed of operation. However, the primary consideration affecting the choice of method is whether the communication path is human <---> machine, or machine <---> machine.

AED

Stream-oriented I/O Transmission. A standard function package, called GIN and GOUT, provides a form of list-directed I/O. Data-directed I/O is not supported by AED.

A form of edit-directed output is provided by the ASEMBL and EXPAND subroutine packages. The RWORD package, or the newer FSM package, provides generalized finite-state scanning logic for processing input--but these are far too complex in simple cases. As a practical matter, there is no edit-directed input as commonly understood.

Record-oriented I/O Transmission. Record-oriented, or 'bulk-I/O', is handled by the IOBCP package. It is both general and low-level, requiring considerable work to use. IOBCP is presently restricted to sequential files. On the

IBM 360 or 370 under OS, a low-level interface package to basic (including nonsequential) I/O macros is provided; this package is expected to be compatible with IOBCP.

In summary, I/O is difficult in AED.

COBOL

All files must be opened prior to reading or writing. This opening does not access the first record; the read statement initiates the actual input process. All data items and records are edited for format spacing and may be edited for content (numeric, alphabetic, alphanumeric, edited numeric, edited alphanumeric).

Stream-oriented I/O Transmission. Stream-oriented input-output data transmission does not exist as such in the COBOL language. A report-writing feature is available which allows the programmer to structure and format pages of the output report. Each report is divided into groups and sequences of items. The structuring allows for referral to an entire report by name, major or minor groups, and elementary items within groups. Report groups may be headings, footings, control, or detail print groups. Report groups may be extended over several lines on a page. The GENERATE statement is used to initiate the report writing, producing a series of report lines or pages with a single statement. This feature may only be used with data described in the REPORT Section of the DATA Division, where control attributes are provided for each report group or item. The report files must be opened (OPEN) prior to any use and closed (CLOSE) before the end of the program.

Record-oriented I/O Transmission. The COBOL language is organized around record-oriented data transmission. All data, except non-contiguous working-storage data, is described as a record or record entry. The language provides the capability to input and/or output a continuous character string as a record; to edit all data types on both input and output; and to redefine a file record format, by multiple record descriptions in the file description section. COBOL requires an OPEN statement for all files to be read (not SORT files). This statement must stipulate the type of file (INPUT, OUTPUT, I-O) and the file names; tape files may be specified with the characteristics REVERSED or NO REWIND. If a file has been described as OPTIONAL, a check is made with the first READ statement, and the AT END condition is forced when the file is not present. An OPEN statement must precede the first READ, SEEK, SUSPEND, or WRITE statements.

Declaratives are used with file processing. These are coded in a special section immediately following the PROCEDURE Division header. In this section, special statements are coded for before and after label processing, error procedures for file processing, and procedures for use before the opening or closing of an input/output unit. SORT files may not be referenced in the USE declaratives section.

JOVIAL

Stream-oriented I/O Transmission. JOVIAL's I/O is basically record-oriented.

Record-oriented I/O Transmission. I/O data in JOVIAL is organized as records in files. Files may be binary or Hollerith type, with records which are collections of bits (binary) or bytes (Hollerith). Files are either rigid (fixed-length records) or varying (varying-length records). Input records may consist of a variable, an array, a table, a table entry, or a contiguous subset of a table's entries; output records may additionally consist of a constant. Records are numbered consecutively beginning with 0 for the first, 1 for the second, and the last record is followed by an endfile element.

Files are activated by statements of the form:

OPEN { INPUT / OUTPUT } <filename> [<record>] $

where the file will be positioned at 0 if <record> is omitted and, if <record> is included, positioned at 1 after reading the first record. Files must be OPENed before they may be input.

Files are deactivated by statements of the form:

SHUT { INPUT / OUTPUT } <filename> [<record>] $

where the <record> will be input or output prior to deactivation, and (for OUTPUT only) an endfile element will be written.

Data is input and output by statements of the form:

{ INPUT / OUTPUT } <filename> <record> $

where input and output are normally sequential. The file may be repositioned either forward or backward (if appropriate to the device) by assignment statements of the form:

POS (<filename>) = <n> $

where <n> is a nonnegative integer; a value of 0, for example, repositions the file at the start of the data. POS (<filename>) is also used in expressions to obtain the current position of a file.

PL/I

Stream-oriented I/O Transmission. List-directed stream-oriented transmission is specified by the key word GET (input) or PUT (output), followed by the key word LIST. List-directed data can be any constant allowable in a PL/I program, except sterling constants. Each item in the stream must be separated by one or more blanks or a comma surrounded by any number of blanks. A null field is specified by two commas separated by any number of blanks and causes the value of the associated item to remain unchanged.

Data-directed transmission is specified by the key word DATA (e.g., DATA (<data list>)). Items that cannot be included in data-directed transmission include: parameters, defined variables, based variables, and on input, subscripted variables. On input, if the stream contains an unknown name in the data list, the NAME condition is raised. If the data list is omitted, it is assumed to contain all names known to the block.

Edit-directed transmission is specified by the key word EDIT and is followed by a format list.

EDIT (<data list>) (<format list>)

On input, the data is considered a continuous string and is converted according to the format list. This differs from list-directed I/O where the data is not considered a continuous stream but rather, items separated by blanks or commas with no editing to take place. On output, each item is converted according to the associated format item and placed in the output stream. Format items include: fixed point (F), floating point (E), complex (C), picture (P), character string (A), bit string (B), column position (COLUMN), line position (LINE), new page (PAGE), line skip (SKIP), remote format list (R), and spacing (X).

Record-oriented I/O Transmission. There are four statements in PL/I that cause record-oriented transmission. They are READ (input), WRITE (output), LOCATE (allocates storage in the input buffer for a based variable and sets a pointer to the location), and REWRITE (causes records to be replaced in an update file). A fifth statement, DELETE, is used to delete records in an update file. Records are stored and retrieved either sequentially (CONSECUTIVE data set organization), on the basis of physical position (REGIONAL), or by use of keys (INDEXED).

C.1.b.2.d Program Structure - Executable Units - Statements - Debugging Statements

Program debugging, the process of verifying the correctness of a program, may constitute a significant percentage of the total time for program development, especially in large systems. Most compilers provide for extensive checking for syntactical errors in the source code. However, the debugging process is greatly facilitated when

debugging statements are defined as an integral part of the language to be executed at object time. Representative of techniques which may be provided for are tracing, loop monitoring, snapshots of memory, and instruction timing. In addition, responses may be specified for such execution time events as underflow, overflow, end-of-file, etc. The implementation process is further facilitated if these options can be selectively eliminated from the operational version of the program without disturbing the program proper.

AED — There are no debugging statements as such in AED.

COBOL — Standard COBOL does not have debugging statements. The DISPLAY verb allows for logical console output and the ACCEPT verb allows for logical console input. Progress is being made toward the addition of the TRACE and EXHIBIT verbs, as well as DEBUG which would provide debugging features, (some of these are presently included in some implementations and standard language extensions).

JOVIAL — All debugging statements must be inserted by the programmer using the standard language; no special provision is made.

However, a number of debugging aids are called for in the JOVIAL compiler specification and are categorized as standard or optional aids. Standard aids include: a source program listing (including compiler-generated error messages); a library source program listing for all code loaded from a library; an object program listing (including all machine locations corresponding to compiler-detected messages); and an environment listing (including

all DIRECT code names and statement, procedure, and variable names). Optional aids include: a set-used listing (including all data references by statement number); a run-time error monitor, which provides an identification of each error and the associated statement number; an alter mode of operation for source deck updating; an alter-update mode; and a grammar-checking mode of operation. Standard aids will be provided at compile time unless suppressed, and optional aids will be suppressed unless requested.

PL/I

In PL/I, debugging statements overlap with error control (C.1.b.2.e). There is currently a debugging PL/I compiler provided by IBM which includes a number of debugging aids and statements. In addition to the conditions discussed in the error control section, some are used as debugging statements. The CHECK condition when raised has no effect on the statement being executed, but displays on file SYSPRINT the name of the variable and its new value. PL/I also has a SNAP option of ON statements which, when executing, displays a list of all active procedures on file SYSPRINT.

C.1.b.2.e Program Structure - Executable Units - Statements - Operating System Interface

Programming languages in support of third-generation computer systems (e.g., multiprogramming, multiprocessing, and time-sharing systems) generally provide the capability to interact with the operating system to request services and/or to specify corrective actions to be taken in the event of an error condition. Such operating system interfaces, for run-time application, are primarily

concerned with specifying the handling of interrupts (i.e., events which cannot be predicted or dealt with in a serial fashion). These include resource management exceptions (e.g., 'out of memory'), the initiation and control of parallel processes (asynchronous tasks), and hardware-associated events, such as endfile conditions, overflow, underflow, and data conversion errors. Additionally, an interface with the operating system occurs when a request is made for service; included in this category are dynamic storage allocation, I/O, and status information. The ability to deal naturally with such events is particularly important for system programming and generally useful in all programming.

AED

Multitasking/Asynchronous Processing. There are no facilities for asynchronous processing in AED.

Dynamic Storage Allocation. A standard feature of AED implementations is a sophisticated 'free storage' package for dynamic storage allocation. This package--not, syntactically, part of the language--permits allocation and freeing of arbitrary size blocks (beads) from hierarchially organized 'zones' or storage pools. The management algorithm may be specified differently for different zones, and may be 'tuned' for improved performance after the initial system build.

Garbage collection, as in PL/I, is not automatic--the user must explicitly FREE beads or zones no longer in use.

The free storage package is not implemented as an interface to the operating system, but as a general facility which might, in some instances, call upon operating-system storage management facilities.

Interrupt Processing/Error Control. No specific facilities for interrupt handling or error control are provided in the language. By convention, packages and individual subroutines are passed the LOC of a procedure (or label) to be invoked in case of error. As with the free storage package, this may or may not imply an interface to the operating system.

Interactive Processing. There are no data communication facilities in the AED language.

COBOL

Multitasking/Asynchronous Processing. Operating system interface is accomplished in standard COBOL via the Asynchronous Processing feature. The reserved words associated with this feature are PROCESS, HOLD, SA (saved area), and USE FOR RANDOM PROCESSING. To specify use, the USE section must contain the procedure; when processing is to be initiated, the PROCESS statement is used. All data modified with this processing must be in a current record area. Data modification works in conjunction with the saved area (SA) file description entries. Each PROCESS execution is called a cycle, which may be completed in random order.

Dynamic Storage Allocation. Dynamic storage allocation as an operating system interface is not available as part of the language. Refer to data descriptions and segmentation processes for similar storage handling.

Interrupt Processing/Error Control. Interrupt processing and error control are features of the language, explicitly and implicitly. File errors and interrupts may be handled

in the DECLARATIVES Section, where procedures are specified for label processing and error processing. The AT END clause also provides an interrupt branch capability. The RERUN feature provides a checkpoint restart capability. The user must designate the medium to receive the data and the method of determining the frequency of checkpoints.

Error control may be accomplished by using and testing status switches, specified as SPECIAL-NAMEs in the environment area; by testing for SIZE ERROR in computations; and by class testing for numerically-defined and alphabetically-defined data items. The INVALID-KEY clause may be used for input-output to provide for user procedures, when the controlling key is erroneous.

Interactive Processing. At this time, the standard language does not specify any data communication features.

JOVIAL

Operating System Interface. There is no provision in JOVIAL for direct communication with the operating system except for the STOP statement, which only provides for return of control to the operating system. The JOVIAL program operates as if nothing else could go on in the system at the same time, either multiprogrammed or multiprocessed. Any support or activity of the operating system of these types is transparent in the JOVIAL program.

PL/I

Multitasking/Asynchronous Processing. PL/I can execute a number of operations concurrently. Each operation or procedure is known as a task. Without multitasking, the plan of control is synchronous; the calling procedure suspends operation until the called procedure returns control to it.

Under multitasking, an additional plan of control is established and both procedures can be executed (in effect) concurrently. The process is known as attaching a task. Any task can attach a number of subtasks (machine-dependent). Each subtask can be given a priority and the major task can question whether a subtask is complete and/or abnormally ended by use of an EVENT option. A WAIT statement can be used to retain activation control until a specified event has been completed.

Dynamic Storage Allocation. Automatic storage is allocated at entry to a block and freed upon exit from the block; static storage persists while the program is loaded.

Storage can be allocated and freed for BASED and CONTROLLED variables (by the user only) through use of the ALLOCATE and FREE statements, respectively. All other storage is allocated and freed without user control.

Interrupt Processing/Error Control. Interrupt processing and error control are accomplished in PL/I by the ON statement, which could also be considered a debug statement (see C.1.b.2.d). The ON statement specifies what action is to be taken when an interrupt results from the occurrence of a specified condition. These conditions include CONVERSION, ENDFILE, SIZE, ZERODIVIDE, UNDEFINED FILE, etc. If SNAP is specified in an ON statement, a calling trace is listed. A REVERT command cancels the effect of the ON statement.

In addition to the ON command discussed above, enabled condition prefixes can be considered a form of error control.

The condition is enclosed in parentheses and prefixed to a statement before the label (i.e., (<condition>): LABEL: <statement>;). If the statement is not a PROCEDURE statement or a BEGIN, it is active only for that statement; before a PROCEDURE statement implies it is active for the entire procedure, and before a BEGIN block statement causes it to be active for the entire BEGIN block. Prefixes are used to specify whether or not interrupts are to be recognized. If the condition appears alone, the interrupts are to occur (e.g., (SIZE):). If the word NO precedes the condition, the interrupt is not to occur (e.g., (NOSIZE):).

Interactive Processing. The TRANSIENT attribute for a file indicates that the contents of a data set are re-established each time the data set is accessed. This means that records can be added to the file by one program, at the same time another program removes records from the file. Therefore, the data set can be considered to be a continuous queue through which records pass in transit between a control program and a processing program.

C.1.b.3 Program Structure - Executable Units - Compound Statements

In the previous section (C.1.b.2) a delineation of executable programming statement types was presented. These included data manipulation statements, program sequence control statements, input/output statements, debugging statements, and operating system interface statements. With minor exceptions, these statements can be classified as simple statements consisting of a single command to accomplish a specific operation.

However, it becomes a programming convenience if such statements can be combined into more complex units which can be named, referenced, and treated logically as single, independent, executable units. Representative of such units are compound statements, loops, blocks, functions, and procedures.

A compound statement, defined as a statement that contains other statements, represents the most elementary combinatorial form. The conditional statement previously discussed is illustrative:

IF <relational expression> THEN <statement-1> ELSE <statement-2>

AED

Compound constructs are provided by:

IF...THEN...ELSE and FOR...DO

where, '...' represents a simple statement. Moreover, a BEGIN...END block may appear wherever a simple statement can, so that nesting to any depth is permitted.

COBOL

Compound statements in COBOL are of three types: file processing, arithmetic, and conditional. Arithmetic compound statements may be written as an imperative statement with a SIZE ERROR condition, followed by an imperative statement.

Compound file processing statements are input/output verbs and clauses followed by an AT END, INVALID-KEY, or END-OF-PAGE, followed by an imperative statement.

Compound conditional statements are written as:

IF <relational expression>

$$\left\{\begin{array}{l}\left\{\begin{array}{l}\text{<imperative statement>}\\ \text{<conditional statement>}\\ \text{<compiler-directive statement>}\end{array}\right\}\left[\begin{array}{l};\text{ELSE <statement>}\\ ;\text{ELSE NEXT SENTENCE}\end{array}\right]\\ \text{NEXT SENTENCE}\end{array}\right.$$

IF-ELSE statements are paired proceeding from left to right. Control passes to the next sentence as written, or to a RETURN statement, or the mechanism of a PERFORM or USE.

JOVIAL

There are four types of compound statements in JOVIAL:

IF <booleanexpression> $ <statement>

IFEITH <booleanexpression> $ <statement>

ORIF <booleanexpression> $ <statement>

FOR <loopexpression> $ <statement>

where the <statement> is necessary to the syntax of the IF, IFEITH, ORIF, or FOR statements. The concluding statement in all of these can also be a BEGIN/END block statement. For more details on these statements, see C.1.b.2.b and C.1.b.4.

PL/I

The only compound statement in PL/I is:

IF (<expression-that-evaluates-to-bit-string>)
THEN <action-if-true>; ELSE <action-if-false>;

The THEN clause is executed if any of the bits in the bit string are 1, and the ELSE clause is executed if all bits are zero.

C.1.b.4 Program Structure - Executable Units - Loops

A loop is a set of statements organized so as to provide for the execution of repetitive operations with varying parameters. It has two essential components, namely, a loop control statement and the loop body proper. Loop control statements, as previously noted (see C.1.b.2.b), typically specify the range of execution, an indexing variable, and the termination criteria. The loop body consists of the statements to be executed iteratively.

Programming languages have specific rules governing the degree to which loops may be nested, the mechanism for incrementing and testing the indexing variable, and the allowable transfers into and out of loops.

AED

The unit controlled by a FOR...DO may be a simple statement or a BEGIN...END group. Branching into and out of loops or other BEGIN...END groups is freely allowed, although branching into the middle of an inactive procedure block is not permissible. Testing for end-of-loop conditions is done at the beginning of each iteration.

COBOL

Internal loops are supported by use of the PERFORM verb with any or all options. These loops may involve one or more paragraphs, but cannot cross sections. The beginning paragraph and the last paragraph to be executed must be specified. The EXIT verb is optional after the last paragraph and must be the only statement in a paragraph.

Another method of looping may be provided by using the ALTER and GO TO DEPENDING ON verbs and options.

Special table looping may be achieved by using the SEARCH verb with indexed table data.

JOVIAL

Loops in JOVIAL are initiated with FOR statements of the form:

FOR <loopvariable> =

{ ALL (<name>)
<expression1>[,<expression2>[,<expression3>]] } $

The FOR statement is followed either by a simple statement or by a BEGIN/END block statement. Any legal executable statement is permitted in conjunction with a FOR statement.

<loopvariable>s are single letter names; either positive or negative incrementation is permitted. Iteration may be terminated at any time from within a loop by statements of the form:

TEST [<loopvariable>] $

where including the <loopvariable> causes control to go directly to the iteration mechanism of that <loopvariable>'s loop (any inner-nested loops are therefore totally terminated), and omitting the <loopvariable> simply causes control to go directly to the iteration mechanism of the nearest loop.

Control must not go from outside a loop to a statement inside a loop without first going through the loop's FOR statement. Control can be passed to PROC procedures outside the loop, if control is returned by normal subroutine termination (i.e., RETURN or parameter exit to a label within the loop); but outside CLOSE procedures may be invoked only if control does not return to the loop.

PL/I

Control may transfer out of a DO loop at any place but control may enter only at the first statement.

C.1.b.5 Program Structure - Executable Units - Blocks/Paragraphs

A block (or paragraph) constitutes a logical processing entity in the hierarchy of executable programming units. Not uniformly defined in all programming languages, such constructs provide the capability to group a set of statements so that they may be treated as the syntactical equivalent of a single statement. A block structure is usually delimited by key words (e.g., BEGIN, END) or a header label, and may contain declaratives to delimit the scope of names. Blocks are not typically called out-of-line; activation is by normal sequential statement flow.

AED

Fundamental statement groupings in AED are the procedure and the BEGIN...END block. The latter may, in general, appear wherever a simple statement might appear. Furthermore, a block may have a value (that of the last assignment within it), if a type name precedes the BEGIN.

COBOL

Paragraphs, the lowest level of labeling, may be executed as procedures with a PERFORM verb and optional clauses. Sections, the next level of labeling, constitute a block of code; these also may be executed as procedures with a PERFORM statement. A special block is the DECLARATIVES which is used for input/output procedures only. The SORT paragraphs (input and output) are closed procedures; i.e., transfers to or from any other main procedure are not allowed.

All paragraphs and sections, except for the DECLARATIVES and the SORT input and output procedures, may be executed in a direct sequential manner or by a procedure-branching PERFORM statement.

JOVIAL

There are two kinds of blocks in JOVIAL: START/TERM, and BEGIN/END.

START/TERM blocks define main programs or external CLOSE routines (i.e., subroutines). Main programs are defined as follows:

START [<numericorigin>] $
<statementlist>
TERM [<statementname>] $

where <numericorigin> establishes the starting point of the program and <statementname> establishes the first statement to be executed on entry. CLOSE routines are defined in the same way as main programs except that START is preceded by:

CLOSE<programname>$

BEGIN/END blocks serve two purposes. The first is to permit the grouping of more than one statement so that a group of statements may take the place of a single statement (in this case, the BEGIN/END block is a sort of superstatement). The second purpose is for internal procedure definitions, both PROC and CLOSE, where the statements representing the procedure are enclosed by BEGIN/END brackets. The scope of data defined in BEGIN/END blocks is not limited to the block unless the block is part of a PROC or CLOSE definition.

PL/I

Sequences of statement are grouped in three ways in PL/I: procedures, groups, and blocks. A group is headed by a DO statement and terminated by a corresponding END statement. A group permits multiple statements to be substituted for a single statement. A block defines an area

of a program. A program may consist of one or more blocks. There are two kinds of blocks: BEGIN and PROCEDURE. Both are terminated by END statements. Every BEGIN block must be contained within a PROCEDURE block. Execution passes sequentially into and out of BEGIN blocks, but procedures must be invoked explicitly. The first (MAIN) procedure is automatically invoked by the operating system. Blocks establish the scope of identifiers.

C.1.b.6 Program Structure - Executable Units - Functions/Built-in Functions

A function is a unique type of subprogram which returns a single computational value whenever it is invoked. Functions must be defined, according to language-specific rules, in terms of the input parameters and the processing logic. Within the subprogram definition, the function name is assigned a value; the subsequent appearance of the function name in an expression generates a call to the subprogram which evaluates the function and returns the computed value and control to the calling program.

Most programming languages provide for the inclusion of a comprehensive set of built-in functions, oriented toward basic operations relevant to the application areas for which the language was designed. The versatility of a language may be greatly enhanced by the existence of such built-in functions, which sometimes provide the semantics of language features not included in the basic language definition.

AED

Functions are simply procedures with an associated type, and consequently an additional implied argument (the returned value). Function references may appear on the left of an assignment in AED, e.g.,

F(X,Y) = Z

which is interpreted as equivalent to the simple reference

F(X,Y,Z).

Since the called procedure F, using ISARG, can determine the number of arguments passed, it can distinguish this 'storing' reference from the 'loading' reference

W = F(X,Y).

This useful feature is a practical aspect of the 'universal reference notation', which allows the actual mechanization of a logical entity (function, array, macro, or bead) to be postponed or changed without affecting the formal references.

A vast number of functions come built into the language or in the standard library. These are generally grouped into packages, the principal packages being a free storage (dynamic allocation) package, a 'string' (data-structure) package, a character-string handling package, and several I/O packages.

COBOL

The language specifications allow for implementation-defined arithmetic functions to be used in conjunction with the COMPUTE verb. This verb provides for arithmetic expressions. Generally SQRT, EXP, and LOG are the only mathematical functions provided. Some implementations have made use of the SPECIAL-NAMES feature to incorporate system functions as name-addressable functions. Special registers are features of the language which provide for automatic counting of line numbers, page numbers, counters, and occurrences of values (TALLY) in an examined character string.

JOVIAL

Most built-in functions ((* and *), (/ and /), ABS, ALL, BIT, BYTE, CHAR, ENT, ENTRY, 'LOC, MANT, NENT, NWDSEN, ODD, POS, REM, REMQUO) in JOVIAL have reserved-word names; the two exceptions are REM and REMQUO. This section gives alphabetically ordered descriptions of each function or function-like element of JOVIAL, beginning with punctuation-defined forms. A discussion of how functions are defined may be found under C.1.a.2.

(* and *) provide for exponentiation of numeric expressions exactly as provided by the operator **; for example, 2 (* 3 *) gives the value 8.

(/ and /) provide for absolute values of numeric expressions exactly as provided by the ABS function; for example, (/ -2 /) gives the value 2.

ABS provides absolute values of numeric expressions. The result is the same data type as the evaluated expression. The general form is:

ABS (<numericexpression>).

ALL is used only in FOR statements to indicate that the <loopvariable> will begin with the number of table entries minus 1 and be decremented by 1 to 0. The general form is:

ALL ({ <tablename> / <tableitemname> }).

BIT is used to substring bit strings from simple or indexed numeric variables (BIT must not be applied to

floating items, however). The result is an unsigned integer, and the general form is:

BIT ($<startbit>[,<numberbits>] $) (<variable>)

where bits are numbered from 0 for the leftmost bit.

BYTE is used to substring byte strings from simple or indexed literal variables. The result is Hollerith or transmission code depending on the variable, and the general form is:

BYTE ($<startbyte>[,<numberbytes>] $) (<variable>)

where bytes are numbered from 0 for the leftmost byte.

CHAR is used to access the characteristic of a floating simple or indexed variable. The result is a negative, positive, or zero integer value; negative values are signed-magnitude form. The general form is:

CHAR (<variablename>).

ENT and ENTRY are used identically to access the full set of values in a table entry. The result has the value 0, if all bits of the entry are 0; otherwise, there is no JOVIAL constant equivalent to the result. The general form is:

{ENT / ENTRY} (<tablename> ($<index>$)).

'LOC is used to make the numeric origin of an item, table, statement, or 'PROGRAM procedure available as an unsigned integer. The general form is:

'LOC (<name>)

where <name> must be followed by a period for statements

and 'PROGRAM names, otherwise not.

MANT is used to access the mantissa of a floating variable (either simple or indexed). The result is a fixed value. The general form is:

MANT (<floatingvariable>)

NENT is used to obtain the number of entries in a table as an integer value. The general form is:

NENT (<tablename>)

NWDSEN is used to obtain the number of machine words per entry of a table. The result is an integer value and the general form is:

NWDSEN (<tablename>)

ODD is used to obtain a Boolean value, which is true (1) if the least significant bit of the parameter is one, and false (0) if the bit is zero. The general form is:

ODD (<variablename>)

where <variablename> may be a simple or indexed numeric variable or a loopvariable.

POS is used to obtain the position value of a file, so that a 0 is returned if the file is positioned at the start of the first data record, and (n-1) if at the nth record. The result is an integer and the general form is:

POS (<filename>)

REM is used to access the remainder of the division of two integers. The result is an integer and the general

form is:

REM (<numerator>, <denominator>)

REMQUO is a procedure rather than a function, but is included here for completeness. REMQUO is used to access both the remainder and quotient of the division of two integers. Both results are integers and the general form is:

REMQUO (<numerator>, <denominator>=<quotient>, <remainder>) $

PL/I

A procedure is considered a function if a specific result is obtained when invoked. The result is included as part of the RETURN statement (e.g., RETURN (expression)) and is returned as the value of the function to the calling location. In PL/I, some functions can be used on the left side of an assignment statement (i.e., as pseudo-variables), in addition to being used as operands in expressions.

PL/I also contains 90 built-in functions and pseudo-variables, (the greatest number of any of the languages considered). The five classes of functions in PL/I are: computational, condition, based storage, multitasking, and miscellaneous. Most of the computational functions can be used with arguments of more than one type.

C.1.b.7 Program Structure - Executable Units - Subroutines/Procedures

A subroutine (or procedure) consists of a self-contained set of instructions necessary to accomplish a well-defined mathematical or logical operation. A subroutine definition has two essential components, namely, (1) a header statement which defines the name, denotes the attributes, and specifies the associated input/output

parameters (if any), and (2) the subroutine body consisting of the declarations and procedural statements. Subroutines are usually defined in terms of formal or dummy parameters, which are replaced by actual parameters when the routine is called. A closed subroutine, in contrast to an open subroutine, is constructed, via linkages, so that code may be localized and invocation from multiple points within the main program does not result in the generation of any in-line code.

Programming languages differ in various aspects of subroutine usage and parameter passage. In particular, some languages allow for multiple entry points, for the nesting of subroutines, and for recursion (i.e., a call by a subroutine to itself). Language-specific rules govern the complexity of the allowable parameters (e.g., variables, expressions, subroutines), and the correspondence which must exist between the parameter list of the invoked procedure and the parameter list of the invoking statement. Parameter passage may be accomplished in three ways, namely, call-by-value, call-by-name, and call-by-address, each of which results in a different mode of subroutine initialization prior to execution.

Subroutines/procedures provide for economy of storage and support the concept of modular/structured programming. Multi-level program implementation is facilitated in that all code at a given level represents the same degree of abstraction.

AED

Procedures are defined by a statement of the form:

DEFINE [<type>] [RECURSIVE] PROCEDURE <name>
[(<args>) WHERE <arglist>] TOBE <statement>

where <statement> is, most often, a BEGIN...END block.

Procedure definitions may be nested within a procedure definition; the scope of their names is limited to the containing procedure. A number of 'outer' procedures may be defined within a 'program'--a compilation unit of the form:

[<name>] BEGIN...END FINI

(The <name> has no function.) All outer procedures have global (loader-known) scope, and any data declared within a program but outside any procedure is common to all the procedures defined in the program.

A call to a procedure is a reference of the form:

<name>(<arglist>).

References on the left are permitted; F(X,Y)=<expression> is interpreted as F(X,Y<expression>) (see C.1.b.6).

The WHERE clause in a procedure definition declares the expected types of the passed arguments. The type, if any, is declared not only in the DEFINE statement but also within the calling procedure, viz:

<type>[RECURSIVE] PROCEDURE <name>

Arguments are not type-checked, but are assumed to match the parameters. Argument-passing is by location.

The standard ISARG routine package permits handling of variable-length parameter lists.

Multiple entries to a procedure are not permitted. The ability to have multiple outer procedures within a program and shared data makes multiple entries unnecessary.

Static (textual) and dynamic (call) nesting is allowed to any depth.

If a procedure is recursive, it must be so mentioned in the DEFINE and in the PROCEDURE declaration for any calling procedure. There is no provision for generally reentrant code; recursion is a relatively easy special case.

COBOL

The standard language provides for three types of procedures or subroutines: internal, external, and internal assembly code.

Internal procedures require no definition and no arguments, since all data is available for any part of the program. These procedures may be executed in two ways, as part of the PERFORM loop and/or as a direct 'next sentence' entry.

External procedures do not require a specific definition statement; however, if data is to be transferred and a USING clause is specified, this data must be defined in both programs in the LINKAGE Section. The number and attributes of the arguments in the called procedure must be identical to the parameters supplied by the calling program. A called program may not be segmented. External procedures may have multiple entry points defined by the ENTRY verb, which may have a USING clause. Called procedures may call other procedures, but none may call the main or calling procedure. Storage used by a called program may be released by the CANCEL statement in the calling program.

Internal assembly-coded procedures are defined by the ENTER statement followed by the assembly language code. Special care must be given to the boundary positions of data used in these procedures.

JOVIAL

The definition of procedures and subroutines is discussed under C.1.a.2; this section will go into broader aspects of their construction.

Procedures may be defined internally with PROC and CLOSE, and externally with CLOSE (START/TERM). Commonly-used definitions may be centralized in libraries for subsequent inclusion in programs.

PROC procedures are referenced by simply writing the procedure name and parameter list. Functions are referenced by including the name and parameter list in an expression. CLOSE procedures (both internal and external) are referenced by GOTO <closename> statements.

Main programs are defined as follows:

```
START [<numericorogin>] $
<statementlist>
TERM $
```

Data definitions may be introduced via the COMPOOL by explicit definitions, or, for simple items, by reference. COMPOOL-defined data is known throughout all internally-compiled procedures of a program. Data defined for a procedure is known only for all procedures internal to that procedure. Data used by procedures can be passed either

as parameters or by scope of name. Functions have only input or no parameters defined, while procedures may have only input, only output, both, or no parameters.

There is no provision for multiple names of procedures representing multiple entry points, nor for recursion or reentry.

PL/I

A procedure is delimited by a PROCEDURE statement and an END statement. A procedure not included in another block is an external procedure and a procedure included in another block is an internal procedure. All external procedures must be defined by their calling procedures (on IBM machines the first procedure is defined by use of the OPTIONS (MAIN) in the PROCEDURE statement). Internal procedures may or may not be declared, according to the compiler. Subroutine procedures are invoked by calling procedures by the use of a CALL statement. A procedure is identical to a function in PL/I, except that a function returns a value in the RETURN statement.

Parameters are associated with the entry point and are put into one-to-one correspondence with the arguments used in a CALL statement.

Procedures may have more than one entry point in PL/I. Procedures may be invoked either at the primary entry point (the procedure statement) or secondary entry points (the entry statement). The entire procedure is said to be active no matter what point of entry is used. An entry point may have the attribute GENERIC which defines a name as a family of entry names, each of which is referred to by the name declared. The proper entry point is selected

based on the attributes of the arguments specified in the procedure reference.

Procedures can be nested and any procedure that is inside any other block is called an internal procedure. A variable is known only in the block in which it was declared unless it was declared EXTERNAL.

Procedures can be both recursive and reentrant, if the procedure is declared to be RECURSIVE.

C.1.c Program Structure - Compile-time Features

Compile-time features consist of that set of statements which convey information to the compiler about actions to be performed before, or coincidental with, the actual compilation process. As such, operations usually involve the program in its original (source) form. Representative of some such operations are:

1. source text manipulation
2. language rule modification
3. initialization of variables
4. inclusion of other languages

Macro-preprocessors represent perhaps the most powerful capability for source text manipulation. In effect, they provide the capability to achieve the self-extendability of a language and also the specialization of a language for particular applications. Most preprocessors are limited to the expansion of macros, but some of the more advanced programming languages provide for a limited set of logic operations.

The ability to access precoded units (such as library routines, macros, and declarations) from a central file and to insert them into individual programs at compile time is a desirable feature of programming languages. Referred to as 'common text insertion', it is a useful aid in reducing coding effort and in standardizing and centralizing the management of common system elements.

Additionally, the nature of some problems makes it extremely useful to have the capability to modify the standard language rules (for example, to override the preset default options). Some programming languages provide compiler directives for this purpose.

The initialization of variables and the inclusion of 'other language' statements within the source code both require an interaction with the compiler.

AED — Source Text Manipulation. There is a macro preprocessor of the 'skeleton' type: sample code of the type to be generated is written, with dummy variables (parameters) to be filled in at call time. A call is of the form:

<macroname> (<arg1>, <arg2>, ...)

The expansion of the macro (i.e., the skeleton code fleshed out by filling in the actual arguments for the parameters) replaces the call, and is itself rescanned for instances of macro calls. This is weaker, though simpler to use, than 'interpreter' type preprocessors such as PL/I's. However, a user exit (procedure call) to an AED-coded subroutine is also allowed. This subroutine can interact with the preprocessor, subject to its interface rules. Almost any operation should be possible this way, although one suspects that such a facility must be complex to use.

Common source code of any kind, including macro definitions, may be stored in a file and called into individual programs by an INSERT <filename>.

In addition, there is a provision for one-time processing of INSERT files into 'item' format, which can be processed by the compiler much more rapidly than raw input text.

There is also an item-level 'synonym' facility, which allows substitution on a token-for-token basis. Tokens may be ordinary identifiers, key words, or punctuation. For example, one may define ';' to be equivalent to '$'. This synonym facility is independent of the macro preprocessor; synonym definitions are recognized and substitution is performed by both the preprocessor and the compiler.

Language Rule Modification. Except for redefining punctuation (see previous section) the compiler cannot be instructed to change its ways--e.g., to alter the defaults.

Initialization. An interpretive PRESET language, AED-like but a subset of the full language in capability, is available for setting up initial values of variables and data structures at compile time. Elementary arithmetic operations are permitted.

Inclusion of Other Languages. Commendably, AED does not allow switching back and forth with other languages within a procedure. Depending on linkage conventions in a particular installation, other languages may be called from

AED and vice versa. Historically, compatibility of this type with FORTRAN is maintained.

COBOL

Source Text Manipulation. Specific preprocessing statements are not part of the language. The COPY and USE verbs and statements are used, respectively, to COPY source text and incorporate it into a program, and to USE specified input/output techniques on program files. The compiler does generate data positioning and alignment as needed prior to data manipulation, thereby handling all mixed mode, format, and usage differences coded by the programmer.

The COPY verb used in the environment area allows for accessing user-supplied and system-supplied library routines.

Language Rule Modification. There is no provision within COBOL for language rule modification.

Initialization. Initialization of data is accomplished by the VALUE clause defined with elementary data items in the DATA Division. Range of data values is specified by the RANGE clause.

Inclusion of Other Languages. Other language processing is included when the ENTER verb is encountered. All necessary language interfaces are generated by the compiler. Control returns to COBOL with the ENTER COBOL statement.

JOVIAL

Source Text Manipulation. JOVIAL has nothing which corresponds to any sort of macro preprocessor, but provisions

are made for common text insertion and for simple text substitution. COMPOOLs contain definitions, and libraries contain procedure code; both mechanisms are accessed through implementation-specified control cards. JOVIAL's common-text insertion feature provides for centralization of commonly-used code and is particularly useful in large-system implementation.

Simple text substitution is provided by statements of the form:

DEFINE<name> ''<text>'' $

where <name> is a legal JOVIAL name and <text> is any combination of symbols, including other DEFINE <name>s. Once DEFINEd, a <name> may be redefined but never undefined.

Language Rule Modification. Unless explicitly defined, JOVIAL processes all simple item references as if they had been declared integer. The MODE statement provides for general assignment of a data type to all undefined simple items, which can be any legal JOVIAL definition and may include presetting the items. The MODE definition applies to all code following it in the source program and may be overridden by another MODE statement. The MODE statement has the general form:

MODE <itemdescription> [P<optionallysignedconstant>] $

Initialization. JOVIAL provides for full compile-time initialization of variables, usually in the form of BEGIN/END blocks for defined variables and of a MODE option for undefined simple items. All data organizations as well as all data types may be initialized.

Inclusion of Other Languages. JOVIAL provides for the insertion of assembler code between the two primitives DIRECT and JOVIAL. Inserted assembler code generally will conform to assembler formatting requirements; in any case, this feature is very implementation-dependent.

PL/I

Source Text Manipulation. The primary compile-time directives in PL/I are the macro facilities. These facilities allow the programmer to alter the source program, at compile time, in the following various ways:

1. change any identifier appearing in the source program
2. indicate which sections of the program are to be compiled
3. incorporate a string of text from a user library into the source program

In most cases the compile-time statements must be preceded by a percent sign (i.e., % DECLARE A CHARACTER;).

Compile time for PL/I is defined in two stages:

1. Preprocessor Stage - Preprocessor statements are executed to alter the source text being scanned. The altered source text is input to the second stage.
2. Processor Stage - Output of first stage is compiled into an executable object program.

Common text can be inserted in a source program by using the preprocessor. This can be accomplished in one of two ways: (1) a character variable could be replaced by a

string of text which would then be compiled as part of the source program, (2) the text could be included by using the % INCLUDE preprocessor command. This command incorporates strings of external text into the source program.

Preprocessor variables can be specified to be either FIXED or CHARACTER. No other attributes can be declared for a preprocessor variable.

Language Rule Modification. The DEFAULT statement allows a user to specify the default attributes to be applied to designated identifiers that require implicit declarations. The DEFAULT statement can specify attributes for:

1. explicitly declared identifiers
2. contextually declared identifiers
3. parameter attributes
4. values returned from function procedures

Initialization. Declarations cannot be preset in PL/I except that they may be included as part of the code by use of the % INCLUDE command. A new IBM optimizing compiler also allows the user the capability of presetting default declarations.

Inclusion of Other Languages. Because of the way PL/I sets up its procedure calls, procedures written in other higher-level languages cannot be called. The exceptions are programs compiled under the IBM Optimizing Compiler, which allows calls to other language subroutines including FORTRAN and COBOL, but not to PL/I subroutines not compiled under the optimizer.

C.2 Program Structure-Organization

Throughout this appendix the subject languages have been defined in terms of basic tokens, data types and organizations, elementary program statements, and mechanisms for constructing higher-level executable units.

It remains in this final section to discuss the program structure; that is, the sequencing rules governing the organization of the program elements and the implication of this arrangement on the scope of data names. Each language has specific rules relative to program structure and the manner in which subunits may be sequenced and combined. Most, however, are based on the concept of a main executive routine and a collection of subordinate routines, nested to any arbitrary level. Each such subprogram operates on a definitive data set and must have the means of accessing the data and determining the associated data attributes. This may be accomplished by global declarations, but efficiency considerations and multi-programmer implementations usually dictate a more localized approach. Language-specific rules governing the placement of declaration statements within a hierarchy of subunits will allow a common data name to be associated with the same or different storage areas throughout program execution. Thus, the range over which a data name is 'known' is under programmer control.

AED — Program Format. Statements or statement groups to be executed in sequence are written in sequence, separated by '$'. Selection of one of two statements (groups) or alteration of the normal dynamic sequence can occur due to compound (C.1.b.3) or GOTO statements (C.1.b.2.b).

The format of the procedure and the program is discussed in C.1.b.7. Declarations of variables may occur anywhere in the block, except for declarations of the parameters, which must appear in the WHERE clause of the procedure definition (C.1.a.2). However, declarations within an included BEGIN-END will apply only over that block.

Scope of Names. Names generally apply over the innermost block in which they are declared, and any blocks contained therein to any depth. The scope of a procedure name is both internal to the procedure and the immediate external environment of the procedure ('outer' procedures become known to the system loader).

COBOL

Program Format. This is a very structured language, so that most structuring violations will create a severe error at compilation. There are two organizational areas--actual input format and program coding sequence.

The actual 'card' input is divided into five distinct parts; the sequence number field, the continuation field, program area-1 starting field, program area-2 starting field, and the identification field. The first and last fields are optional; the continuation field is used for the line comment code and a continuation symbol (-), needed when an identifier or literal has been split in the middle; program area-1 starting field is used for all headers, section names, paragraph names, file descriptions, non-contiguous data level numbers, and record number indicators (01); program area-2 is used for all sentences and statements. Data description entries (other than 01 and 77) may

start anywhere after the start of program area-1. All sentences and clauses must be in area-2 only; if the line is a continuation or a new sentence, the allocate area-1 field must be blank. The actual size of each area is an implementation design option.

The COBOL program itself must be organized in a standard set sequence. The basic sequence is division, section, paragraph, sentence, statement, and clause. The specific program requirements are:

1. IDENTIFICATION DIVISION. The mandatory paragraphs are the PROGRAM-ID paragraph with 8-character names, and DATE-COMPILED paragraph with a user-supplied date. Other optional paragraphs are AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY.
2. ENVIRONMENT DIVISION. This division has two sections: CONFIGURATION and INPUT-OUTPUT. The first section, CONFIGURATION SECTION, has two mandatory paragraphs: SOURCE-COMPUTER to specify the compiling equipment, and OBJECT-COMPUTER to specify the executing equipment. The optional paragraph is SPECIAL-NAMES. The second section, INPUT-OUTPUT SECTION, has two major paragraphs: FILE-CONTROL which names the files and the external media with hardware attributes, and I-O-CONTROL which specifies the memory area shared, the processing techniques, file location on multiple-file tapes, and the rerun points.
3. The DATA DIVISION is divided into optional sections: FILE, WORKING-STORAGE, CONSTANT, LINKAGE, and REPORT. The division header may have a PREPARED FOR clause, if the data descriptions were written for a

computer other than the object computer. The sections, even though optional, must appear in the following sequence: FILE SECTION, WORKING-STORAGE SECTION, CONSTANT SECTION, LINKAGE SECTION, and REPORT SECTION.

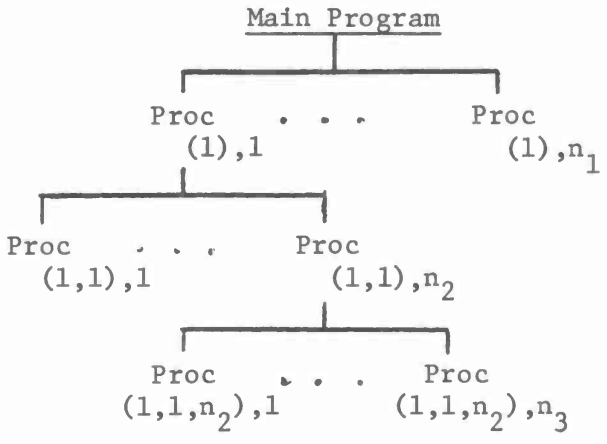
4. The PROCEDURE DIVISION must be included in every program. The DECLARATIVE Section must be at the beginning of this division, started by the reserved word DECLARATIVES and terminated by END DECLARATIVES. The remainder of this division is composed of sections and paragraphs named by the user. If one paragraph is in a named section, than all paragraphs must be in sections. Section names must be unique and paragraph names must be unique within a section. The PROCEDURE Division, if in a called program, may have a USING clause in its header, which specifies the arguments being passed.

The physical program ends with the last COBOL sentence in the PROCEDURE Division.

Scope of Names. The data element names are known to the entire internal program because of the basic COBOL structure. Paragraph names are known only within a section, unless qualified. All names must be uniquely identifiable.

JOVIAL

Program Format. The overall organization is based on that of a procedure, which may contain other procedures containing other procedures, etc.; so that, by proper insertion, a hierarchy of structures may be established as follows:



Programs are compiled either as main programs or CLOSE subroutines (i.e., subroutines without parameters, which are accessed with GOTOs and return control to the next statement after the calling GOTO). Programs consist of directives, definitions, and statements. Directives consist of the MODE and DEFINE statements (see C.1.c). Definitions, which may be provided as part of the original source code or included from COMPOOLs and libraries, establish characteristics of elements (i.e., data and internal and external procedures), except for simple items. The definition of an element must precede its use. The general form of a main program is:

START [<numericorigin>] $
<statementlist>
TERM [<entrystatementlabel>] $

The general form of a CLOSE subroutine is:

CLOSE<closename>$
START [<numericorogin>] $
<statementlist>
TERM $

Scope of Names. In terms of the hierarchy shown above, names generally are known only in those procedures subordinate to the one in which the name is most recently defined. COMPOOL and library definitions always occur at the highest level; all names defined at the highest level are said to have global scope, and all others have local scope. The definition of a name at a higher level may be overridden by redefining the name at a lower subordinate level. Each loopvariable name is local to its loop, to all nested loops, and to all procedures called from these loops. Device names are predefined for each implementation and so are global in scope. The same name may be used without conflict for (1) a device, (2) a statement, program (i.e., PROGRAM or CLOSE), or switch, and (3) an item, table, file, procedure, or function.

The only names which do not need to be defined before being used are simple item names (which then default to integer or the most recent MODE definitions), statement names, functional modifiers, and REM and REMQUO.

PL/I

Program Format. There are no sequence rules required for PL/I except those dictated by the program logic. Declarations, internal procedures, and remote formats may appear anywhere and, unless altered, control flows through blocks sequentially. However, preprocessor data must be defined before it is used.

Scope of Names. The scope of a name is dependent upon its type of declaration. There are three types of declarations in PL/I: explicit, contextual, and implicit. If a variable

is declared explicitly, the scope of its name is the block to which it is internal, except those blocks to which another explicit declaration of the same variable is internal. A variable declared contextually has a scope as if the variable were explicitly declared following the PROCEDURE statement of the external procedure in which the name appears. This has the same effect as if the variable were declared in the external procedure, even if the statement causing the contextual declaration appeared internal to a block contained in the external procedure. Finally, the scope of implicit declarations is as if the variable were explicitly declared after the first PROCEDURE statement of the external procedure in which the name is used.

Variables may also be declared with the EXTERNAL attribute in more than one external procedure or block. Each declaration establishes a scope and the scopes are linked together so that all such references refer to the same name. The scope of the name is the sum of the scopes of all the declarations of that name within the program.

BIBLIOGRAPHY

Language Reference Manuals

AED — AED-0 Programmers Guide, Report ESL-R-406, M.I.T., Electronic Systems Laboratory, Cambridge, Massachusetts, January 1970.

COBOL — Codasyl COBOL Journal of Development 1968, National Bureau of Standards Handbook 106, U. S. Government Printing Office, Washington, D. C., July 1969.

JOVIAL — Standard Computer Programming Language for Air Force Command and Control System (CED 2400), Air Force Manual AFM 100-24, Department of the Air Force, Washington, D. C., June 1967.

PL/I — IBM System/360 Operating System PL/I (F) Language Reference Manual, GC28-8201-3, IBM Corporation, White Plains, New York, July 1970.

Texts

Galler, B. A., and Perlis, A. J., A View of Programming Languages, Addison-Wesley Publishing Company, Reading, Massachusetts, 1970.

Hassitt, A., Computer Programming and Computer Systems, Academic Press, New York, N. Y., 1967.

Hellerman, H., Digital Computer System Principles, McGraw-Hill Book Company, New York, N. Y., 1967.

Higman, B., A Comparative Study of Programming Languages, American Elsevier Publishing Co., Inc., New York, N. Y., 1967.

Johnson, L. R., System Structure in Data, Programs, and Computers, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1970.

Rosen, S., Programming Systems and Languages, McGraw-Hill Book Company, New York, N. Y., 1967.

Sammet, J. E., Programming Languages: History and Fundamentals, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1969.

Sanderson, P. C., Computer Languages: A Practical Guide to the Chief Programming Languages, Philosophical Library Inc., New York, N. Y., 1970.

Texts (Concluded)

Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill Book Company, New York, N. Y., 1968.

Reports and Articles

Ayers, E. R., "The Current State of COBOL", Software 71, Proceedings of a Conference Sponsored by Software World, Maxwell Scientific International, Inc., Fairview Park, Elmsford, New York, 1971, pp. 127-135.

Beech, D., "A Structural View of PL/I", Computing Surveys, Volume 2, No. 1, March 1970, pp. 33-64.

Bemer, R. W., "A View of the History of COBOL", Honeywell Computer Journal, Volume 5, No. 3, 1971, pp. 130-135.

Brampton, M. N., "PL/I Compared with Other Major Languages", Software 71, Proceedings of a Conference Sponsored by Software World, Maxwell Scientific International, Inc., Fairview Park, Elmsford, New York, 1971, pp. 141-143.

Bromberg, H., "The COBOL Conclusion", Datamation, Volume 13, No. 3, March 1967, pp. 45-50.

Brooks, F. P., Jr., "Programming Systems and Programming Languages", Proceedings Third Australian Computer Conference, Australian Trade Publication Pty. Ltd., Chippendale, N. S. W., Australia, May 1966, pp. 265-269.

Budd, A. E., A Method for the Evaluation of Software: Procedural Language Compilers - Particularly COBOL and FORTRAN, The MITRE Corporation, MTR 197, Volume 2, Bedford, Massachusetts, April 1966.

Burkhardt, W. H., "PL/I: An Evaluation", Datamation, Volume 12, No. 11, November 1966, pp. 31-39.

Callender, E. D., and Rhodus, N. W., J-3, PL/I and a Data Base, Aerospace Corp., San Bernadino, California, February 1969. Available from Defense Documentation Center as SAMSO-TR-69-25 (AD682305).

Chapin, N., "What Choice of Programming Languages?" Computers and Automation, Volume 14, No. 2., February 1965, pp. 12-14.

Corbató, F. J., "PL/I As a Tool for System Programming", Datamation, Volume 15, No. 5, May 1969, pp. 68-76.

Currie, R. L., "PL/I Compared with ALGOL, COBOL, and FORTRAN", Proceedings Third Australian Computer Conference, Australia Trade Publication Pty. Ltd., Chippendale, N. S. W., Australia, May 1966, pp. 377-379.

DeBlasi, J., "COBOL Versus UNCOBOL", Datamation, Volume 14, No. 6, June 1968, pp. 67-69.

Dorn, P. H., "Why Another Programming Language?", Data Processing Volume XIII, Proceedings of the 1968 International Data Processing Conference and Business Exposition, Data Processing Management Association, Washington, D. C., June 1968, pp. 68-76.

Dos, K. and Otto, H., "Optimal Dynamic Use of Memory for PL/I Object Programs in a Real Memory Environment", The Computer Journal, Volume 15, No. 1, February 1972, pp. 18-20.

Edelman, H., "A Short Guide to the Wonderful World of COBOL", Datamation, Volume 15, No. 12, December 1969, pp. 161-164.

Engel, F. Jr., The Air Force JOVIAL Compiler Validation System (JCVS), The MITRE Corporation, MTR 2091, Bedford, Massachusetts, April 1971.

Freiburghouse, R. A., "The Multics PL/I Compiler", Proceedings AFIPS 1969 Fall Joint Computer Conference, Volume 35, pp. 187-199.

Gauthier, R. L., "PL/I - Pros and Cons", Data Processing Volume XIII, Proceedings of the 1968 International Data Processing Conference and Business Exposition, Data Processing Management Association, Washington, D. C., June 1968, pp. 76-81.

Gauthier, R. L., "PL/I Compile Time Facilities", Datamation, Volume 14, No. 12, December 1968, pp. 32-34.

Giles, P., "Mini-COBOL", The Computer Journal, Volume 12, No. 3, August 1969, pp. 208-214.

Haverty, J. F., Programming Language Selection for Command and Control Applications, The RAND Corp., P-2967, September 1964.

Hess, H., and Martin, C., "TACPOL - A Tactical C&C Subset of PL/I", Datamation, Volume 16, No. 4, April 1970, pp. 151-157.

Hicks, H. T., Jr., "Modular Programming in COBOL", Datamation, Volume 14, No. 5, May 1968, pp. 50-59.

Hicks, H. T., Jr., "The Air Force COBOL Compiler Validation System", Datamation, Volume 15, No. 8, August 1969, pp. 73-81.

Hicks, H. T., Jr., "A Communication Facility for COBOL", Datamation, Volume 15, No. 12, December 1969, pp. 148-158.

Hicks, H. T., Jr., "ANSI COBOL", Datamation, Volume 16, No. 14, November 1970, pp. 32-36.

Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T., "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques", Communications of the ACM, Volume 11, No. 12, December 1968, pp. 805-813.

Lawson, H. W., Jr., "PL/I List Processing", Communications of the ACM, Volume 10, No. 6, June 1967, pp. 358-367.

McCracken, D. D., "The New Programming Language", Datamation, Volume 10, No. 7, July 1964, pp. 31-36.

Naftaly, S. M.,"Compiling a COBOL Questionnaire", Datamation, Volume 10, No. 8, August 1964, pp. 30-33.

Nicholls, J. E., "PL/I - A Status Report", Software 71 - Proceedings of a Conference Sponsored by Software World, Maxwell Scientific International, Inc., Fairview Park, Elmsford, New York, 1971, pp. 120-126.

O'Brien, W. M., Jovial Evaluation Project, Data Dynamics, Inc., Los Angeles, California, October 1968. Available from Defense Documentation Center, Alexandria, Virginia, as ESD-TR-68-452.

O'Brien, W. M., Jovial Application Questionnaire, Data Dynamics, Inc., Los Angeles, California, December 1968. Available from Defense Documentation Center, Alexandria, Virginia, as ESD-TR-68-454 (AD 681471).

O'Brien, W. M., Approach for Change - Jovial Evaluation Project, Data Dyanmics, Inc., Los Angeles, California, December 1968. Available from Defense Documentation Center, Alexandria, Virginia, as ESD-TR-68-455 (AD681472).

Perstein, M. H., Some Techniques for Describing Programming Languages, System Development Corp. SP-2916/000/01, Santa Monica, California, January 1968.

Radin, G., and Rogoway, H. P., "NPL: Highlights of a New Programming Language", Communications of the ACM, Volume 8, No. 1, January 1965, pp. 9-17.

Raphael, B., "The Structure of Programming Languages", Communications of the ACM, Volume 9, No. 2, February 1966, pp. 67-71.

Remy, E. H., "Learning to Use PL/I", Datamation, Volume 16, No. 7, July 1970, pp. 47-51.

Ross, D. T., "The AED Approach to Generalized Computer-Aided Design", Proceedings 22nd National Conference ACM, 1967, pp. 367-385.

Ross, D. T., "The AED Free Storage Package", Communications of the ACM, Volume 10, No. 8, August 1967, pp. 481-491.

Ross, D. T., and Brackett, J. W., "Automated Engineering Design (AED) Used for Graphics", Honeywell Computer Journal, Volume 5, No. 3, 1971, pp. 136-139.

Ross, D. T., and Rodriguez, J. E., "Theoretical Foundations for the Computer-Aided Design System", Proceedings AFIPS 1963 Spring Joint Computer Conference, Volume 23, pp. 305-322.

Rubey, R. J., "A Comparative Evaluation of PL/I", Datamation, Volume 14, No. 12, December 1968, pp. 22-25.

Rubey, R. J., et al, Comparative Evaluation of PL/I, Logicon, Inc., San Pedro, California, April 1968. Available from the National Technical Information Service, Springfield, Virginia, as ESD-TR-68-150.

Sammet, J. E., "Programming Languages: Current and Future Trends", Computers and Automation, Volume 16, No. 3, March 1967, pp. 32-34, 38.

Sanderson, J. G., "The Theory of Programming Languages - A Survey", Proceedings Third Australian Computer Conference, Australia Trade Publication Pty, Ltd., Chippendale, N.S.W., Australia, May 1966, pp. 321-324.

Schwartz, J. I., "Comparing Programming Languages", Computers and Automation, Volume 14, No. 2, February 1965, pp. 15-16, 26.

Shaw, C. J., An Outline/Questionnaire for Describing and Evaluating Procedure-Oriented Programming Languages and Their Compilers, System Development Corp., FN-6821/000/00, Santa Monica, California, August 1962.

Shaw, C. J., "JOVIAL and Its Documentation", Communications of the ACM, Volume 6, No. 3, March 1963, pp. 89-91.

Shaw, C. J., "A Specification of JOVIAL", Communications of the ACM, Volume 6, No. 12, December 1963, pp. 721-736.

Shaw, C. J., "A Comparative Evaluation of JOVIAL and FORTRAN IV", Automatic Programming Information, No. 22, College of Technology, Brighton, England, August 1964.

Shaw, C. J., "PL/I for C&C?", Datamation, Volume 14, No. 12, December 1968, pp. 26-31.

Sibley, R. A., "A New Programming Language: PL/I", Proceedings 20th National Conference ACM, 1965, pp. 543-563.

Sullivan, J. E., A Comparison of the Programming Languages PL/I and AED Applied to Text Processing, The MITRE Corporation, MTR 2221, Bedford, Massachusetts, September 1971.

Vaughn, P. H., "Can COBOL Cope?", Datamation, Volume 16, No. 10, September 1970, pp. 42-46.

Wexelblat, R. L., "History of the PL/I Programming Language", Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems, Princeton University, Princeton, New Jersey, March 1971, pp. 171-181.

Wigg, J. D., "CCBOL Coding Standards", The Computer Bulletin, Volume 15, No. 7, July 1971, pp. 249-251.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
The MITRE Corporation P.O. Box 208 Bedford, Mass.	UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

A GENERAL BASIS FOR COMPARATIVE EVALUATION OF AED, COBOL, JOVIAL, AND PL/1

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

5. AUTHOR(S) (First name, middle initial, last name)

J.C. DESROCHES

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
FEBRUARY 1973	193	71

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F19628-71-C-0002	ESD-TR-72-330
b. PROJECT NO. 572B	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	MTR-2429

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Deputy for Command and Management Systems Electronic Systems Division, AFSC L.G. Hanscom Field, Bedford, Mass.

13. ABSTRACT

This report provides an analysis of the technical features and pertinent characteristics of the programming languages AED, COBOL, JOVIAL, and PL/1, which were chosen for evaluation because of general applicability to programming problems within the scope of Air Force interest. The methodology derives from the development of a Language Feature Outline and a Language Evaluation Questionnaire for which programmer/analysts supplied detailed technical information and subjective evaluations. The intent of this report is to provide material in support of evaluations of the relative suitability of the four languages for specific applications.

DD FORM 1 NOV 65 1473

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
AED						
COBOL						
JOVIAL						
LANGUAGE STRUCTURE						
PL/1						