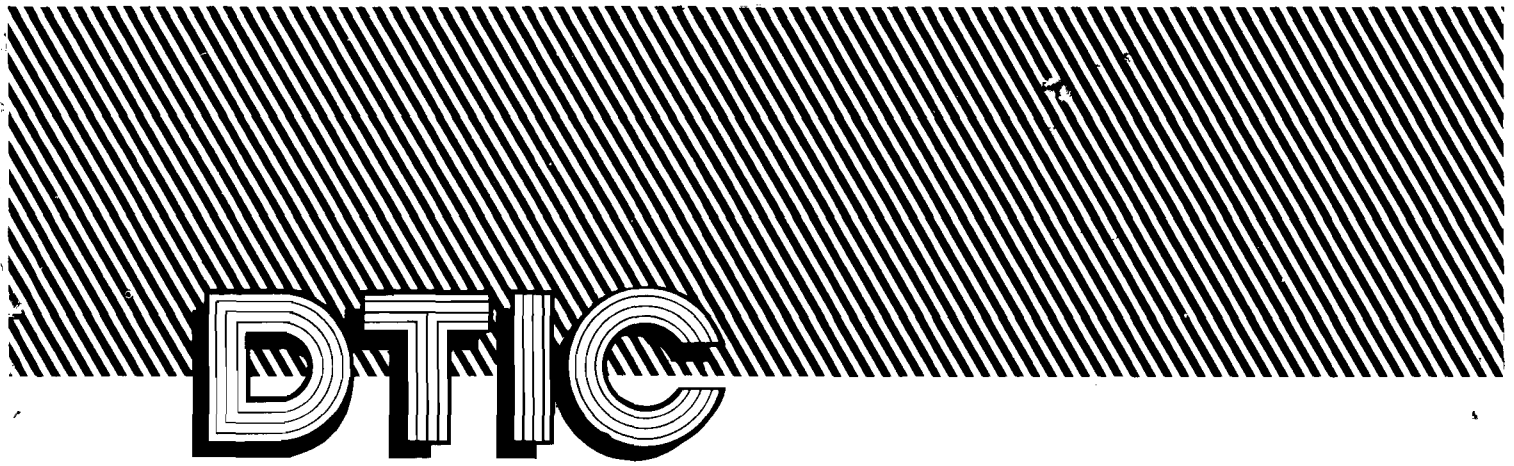


Technical Report

distributed by

Defense Technical Information Center
DEFENSE LOGISTICS AGENCY
Cameron Station • Alexandria, Virginia 22314

A Common Programming Language for the Department of Defense Background & Technical Requirements

Institute for Defense Analyses

Prepared For
Defense Advanced Res. Projects Agency

June 1976

KEEP UP TO DATE

Between the time you ordered this report—which is only one of the hundreds of thousands in the NTIS information collection available to you—and the time you are reading this message, several *new* reports relevant to your interests probably have entered the collection.

Subscribe to the **Weekly Government Abstracts** series that will bring you summaries of new reports *as soon as they are received by NTIS* from the originators of the research. The WGA's are an NTIS weekly newsletter service covering the most recent research findings in 25 areas of industrial, technological, and sociological interest—invaluable information for executives and professionals who must keep up to date.

The executive and professional information service provided by NTIS in the **Weekly Government Abstracts** newsletters will give you thorough and comprehensive coverage of government-conducted or sponsored research activities. And you'll get this important information within two weeks of the time it's released by originating agencies.

WGA newsletters are computer produced and electronically photocomposed to slash the time gap between the release of a report and its availability. You can learn about technical innovations immediately—and use them in the most meaningful and productive ways possible for your organization. Please request NTIS-PR-205/PCW for more information.

The weekly newsletter series will keep you current. But *learn what you have missed in the past* by ordering a computer **NTISearch** of all the research reports in your area of interest, dating as far back as 1964, if you wish. Please request NTIS-PR-186/PCN for more information.

WRITE: Managing Editor
5285 Port Royal Road
Springfield, VA 22161

Keep Up To Date With SRIM

SRIM (Selected Research in Microfiche) provides you with regular, automatic distribution of the complete texts of NTIS research reports *only* in the subject areas you select. SRIM covers almost all Government research reports by subject area and/or the originating Federal or local government agency. You may subscribe by any category or subcategory of our WGA **(Weekly Government Abstracts)** or **Government Reports Announcements and Index** categories, or to the reports issued by a particular agency such as the Department of Defense, Federal Energy Administration, or Environmental Protection Agency. Other options that will give you greater selectivity are available on request.

The cost of SRIM service is only 45¢ domestic (60¢ foreign) for each complete microfiched report. Your SRIM service begins as soon as your order is received and processed and you will receive biweekly shipments thereafter. If you wish, your service will be backdated to furnish you microfiche of reports issued earlier.

Because of contractual arrangements with several Special Technology Groups, not all NTIS reports are distributed in the SRIM program. You will receive a notice in your microfiche shipments identifying the exceptionally priced reports not available through SRIM.

A deposit account with NTIS is required before this service can be initiated. If you have specific questions concerning this service, please call (703) 451-1558, or write NTIS, attention SRIM Product Manager.

This information product distributed by

NTIS U.S. DEPARTMENT OF COMMERCE
National Technical Information Service
5285 Port Royal Road
Springfield, Virginia 22161

PAPER P-1191

A COMMON PROGRAMMING LANGUAGE FOR THE DEPARTMENT OF DEFENSE -- BACKGROUND AND TECHNICAL REQUIREMENTS

D. A. Fisher

June 1976

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION

IDA Log No. HQ 76-18215
Copy 126 of 155 copies

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Paper P-1191	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Common Programming Language for the Department of Defense—Background and Technical Requirements		5. TYPE OF REPORT & PERIOD COVERED Final January-December 1975 6. PERFORMING ORG. REPORT NUMBER P-1191
7. AUTHOR(s) D.A. Fisher		8. CONTRACT OR GRANT NUMBER(s) DAHC15 73 C 0200
9. PERFORMING ORGANIZATION NAME AND ADDRESS INSTITUTE FOR DEFENSE ANALYSES 400 Army-Navy Drive Arlington, Virginia 22202		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task T-36
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE June 1976 13. NUMBER OF PAGES 158 152
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ODDR&E, Electronics & Physical Sciences		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

None

18. SUPPLEMENTARY NOTES

N/A

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

Embedded Computer Systems (Common), Programming Languages, Software Commonality, Real time, Compilers, Software tools, Data types, Machine representations, Translators, Control Structures, Syntax, Programming Language Semantics, Programming Language Design Criteria

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

This paper presents the set of characteristics needed for a common programming language of embedded computer systems applications in the DoD. In addition, it describes the background, purpose, and organization of the DoD Common Programming Language Efforts. It reviews the issues considered in developing the needed language characteristics, explains how certain trade-offs and potential conflicts were resolved, and discusses the criteria used to ensure that any language satisfying the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

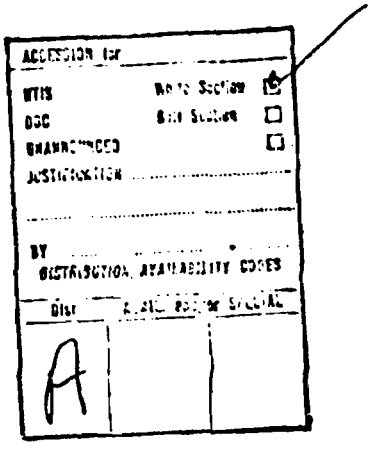
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

PRICES SUBJECT TO CHANGE

20.

criteria will be suitable for embedded computer applications, will not aggravate existing software problems, and will be suitable for standardization.



PAPER P-1191

A COMMON PROGRAMMING LANGUAGE FOR THE DEPARTMENT OF DEFENSE - - BACKGROUND AND TECHNICAL REQUIREMENTS

D. A. Fisher

June 1976

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION
400 Army-Navy Drive, Arlington, Virginia 22202

Contract DAHC15 73 C 0200
Task T-36

ABSTRACT

This paper presents the set of characteristics needed for a common programming language of embedded computer systems applications in the DoD. In addition, it describes the background, purpose, and organization of the DoD Common Programming Language efforts. It reviews the issues considered in developing the needed language characteristics, explains how certain trade-offs and potential conflicts were resolved, and discusses the criteria used to ensure that any language satisfying the criteria will be suitable for embedded computer applications, will not aggravate existing software problems, and will be suitable for standardization.

ACKNOWLEDGMENT

The author is pleased to acknowledge the many valuable contributions and comments from individuals and organizations inside and outside the Department of Defense. A list of contributing organizations and individuals is given as the Appendix, with apologies to those whose names may have been inadvertently omitted. Special thanks are deserved by those who took opposing positions and, thereby, exposed fundamental issues.

The author is also indebted to Thomas A. Standish and John B. Goodenough for their reviews and many valuable comments and suggestions on this paper.

PREFACE

This paper was prepared for the Office of the Director of Defense Research and Engineering (Electronics and Physical Sciences) as Part 1, Software Research and Development, of Task T-36 (revised), "Evaluations of Options in Electronic Technology". Task T-36 provides independent evaluations of selected areas of electronic technology where the Services are pursuing different technical approaches to similar problems.

Portions of this document have appeared in "Programming Language Commonality in the Department of Defense", by D. A. Fisher, *Defense Management Journal*, Vol. 11, No. 4, (October (1975), pp. 29-33.

CONTENTS

SUMMARY

This document, which reports the work of the author in support of the DoD Higher Order Language Working Group, is intended to provide the Services with the necessary technical guidelines to achieve their goal of programming language commonality for embedded computer applications in the Department of Defense.* It provides background on the software and programming language problems in the DoD, presents the language design/selection criteria used to guide evaluation of technical characteristics, and identifies the characteristics needed for the common language.

The IDA effort provided the background, analysis, and evaluations necessary to reconcile the diverse and sometimes conflicting perceived needs. It included examination of the purpose and expectations for the Higher Order language effort, review of several technical and managerial issues in selecting a common programming language, and analysis of some important trade-offs in the design/selection criteria and in the choice of language characteristics. The selected choices were subjected to intensive critical review by the language's potential users and others concerned, in an attempt to illuminate the issues in a comprehensive way. This document represents the degree to which this has been done.

*An embedded computer system is physically incorporated into a larger system whose primary function is not data processing (*e.g.*, electromechanical system combat weapon system, tactical system, aircraft, ship, missile, spacecraft, command, control, and communication systems) is integral to that system from a design, procurement, and operations viewpoint, and generally includes information, control signals and computer data in its output.

A. BACKGROUND

The problems of digital computer software are complex and poorly understood. Although there are many widely recognized symptoms, the underlying problems are not well delineated and there are few useful quantitative measures for assessing either the importance of perceived problems or the effectiveness of proposed solutions.

1. The DoD Software Problem

Some important software-related problems are listed below. Each item describes a class of unrealized expectations about the development or maintenance of DoD software. These "problems" are unique neither to software nor to the military, but unlike electronic equipment, software has no inherent physical constraints to limit expectations.

- Responsiveness. Computer-based systems often do not meet user needs. This may reflect poor specification of requirements, poor system performance, or lack of flexibility in the software.
- Reliability. Software often fails. Both the probability of software faults and errors and the effects of such errors on system operation must be reduced.
- Cost. Software costs are seldom predictable and are often perceived as excessive. Life-cycle costs are given insufficient consideration during software development.
- Modifiability. Software maintenance is complex, costly, and error prone, and the difficulty in modifying software increases the need for duplicative software development.
- Timeliness. Software is often late and frequently delivered with less-than-promised capability. There are no accurate methods for predicting software production times.
- Transferability. Software from one system is seldom used in another, even when similar functions are required.

- Efficiency. Software development efforts do not make optimal use of the resources (processing time and memory space) involved, especially in embedded computer applications with their real time constraints and often limited hardware resources.

Although the above list is consistent with the findings of many DoD in-house and contractor studies of the software problem (Ref. 1), its elements represent only perceptions of the problem, and, in most cases, are not or cannot be substantiated by quantitative data. For example, software costs are thought to be excessive, but actual software costs are largely unknown and there is little evidence that they can be reduced.

Obvious solutions are not necessarily the best. Efficiency is important, and although any computer program can be rewritten to run faster or to use less memory space, more optimal coding may, in fact, result in higher total costs. There is evidence that software costs grow exponentially with attempts to increase hardware utilization, while hardware costs for increased speed or memory capacity grow linearly, or less. Thus, if the physical constraints on the hardware can be met, the least costly solutions may lie with more capable but underutilized hardware.

2. Character of The DoD Software Environment

Software is becoming increasingly costly to the DoD. Digital computer software costs in the DoD in 1973 were estimated (Ref. 2) at $3 billion to $3.5 billion annually. Between 1968 and 1973, there was a 51 percent increase in total direct cost of DoD computer systems (including both hardware and software) reported under the Brooks Bill (Public Law 89-306, October 1965). These increases occurred even though there were drastic reductions in both unit and total costs of computer hardware and fewer systems were reported in 1973. The increased costs of computer software may reflect a combination of factors, including (a) the

trend toward more automation and increased use of computers, (b) the greater complexity of software resulting from increased expectations and expanded requirements generated by improved hardware and software technology, and (c) rising personnel costs.

The major problems of DoD software are associated with embedded computer systems. Embedded computer system software includes all software which is integral to a larger military system or weapon, including tactical weapons systems, communications, command and control, avionics, simulation, test equipment, training, and systems programming applications. It also includes any software which supports the design, development, or maintenance of such systems. As a general rule, embedded computer software is software for any DoD computer hardware which is not reported under the General Management Category of the Brooks Bill. DoD software which is not in the embedded computer software category is used primarily in data processing and scientific applications.

The majority of software costs in the DoD are associated with embedded computer systems (see Fig. 1). Embedded computer software often is large (50,000 to 100,000 lines and greater), is long-lived (10 to 15 years), is subject to continuing change (annual revisions of the same magnitude as the original software size), and must conform to physical and real time constraints of the associated system hardware and requirements. Scientific applications require the largest and most visible computers in DoD and may use a significant portion of the total computing power, but they represent only about five percent of software costs.

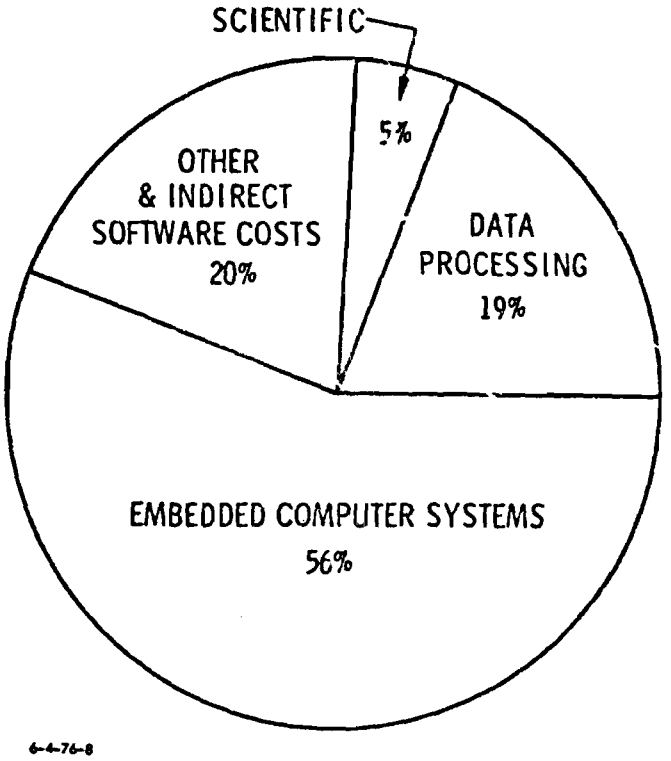


FIGURE 1. Breakdown of Estimated $3 Billion Annual DoD Computer Software Costs [Derived from figures in CCIP-85 and in P-1046 (Refs. 1 and 2)]

3. Programming Languages in The DoD

There are at least 450 general-purpose languages and dialects currently used in the DoD, but it is not known whether the actual number is 500 or 1500. With few exceptions, the only languages used in data processing and scientific applications are, respectively, COBOL and FORTRAN. A larger number of programming languages are used in embedded computer systems applications. The continued proliferation of programming languages for embedded computer software may reflect an unfounded optimism that software problems would disappear if only there were a language better suited to the task at hand. However, the little available evidence indicates that the major payoffs will come from better programming methods and techniques, more software commonality, and more useful and easily accessible software tools and aids.

There are a number of widely held perceptions about the ill effects of the lack of programming language commonality in the DoD. Although these ill effects can be substantiated only by examples, and their true extent is unknown, they have provided much of the incentive for the common-language effort. The lack of programming language commonality in DoD embedded computer applications may:

- Require duplication in training and maintenance for the languages, their compilers, their associated software support packages, and of all the common functions needed in the application.
- Minimize communication among software practitioners and retard technology transfer.
- Result in support software being project-unique and tie software maintenance to the original developer.
- Diffuse expenditures for support and maintenance software so only the most primitive software aids are developed, but repeatedly.
- Limit the applicability of new support software and techniques.

- Create a situation in which the adoption of an existing language by a new project is often more risky and less cost-effective (at least during development) than developing a new programming language specialized to the project.

On the other hand, programming languages are the primary means of introducing new programming methods, tools, techniques, and greater automation into software development and maintenance processes. Consequently, there should be periodic review of the common language(s) for possible upgrading or replacement to accommodate demonstrable and useful advances in software technology and methods. Also, there is no practical way to reimplement existing software, so even if all language proliferation were stopped, it would be 10 to 15 years before the existing languages could be dropped.

B. THE COMMON PROGRAMMING LANGUAGE EFFORT OF THE DoD

1. Background

During 1974, elements in each of the Military Departments independently proposed the adoption of a common programming language for use in the development of major defense systems within their own departments and undertook efforts to achieve that goal. Those efforts included the Army "Implementation Language for Real-Time Systems" study, the Navy CS-4 effort, and the Air Force "High Order Language Standardization for the Air Force" study.

In January 1975, the Director, Defense Research and Engineering (DDR&E), in a memo to the Assistant Secretaries of the Military Departments for R&D, noted the multiple benefits of a single common language for military applications (Ref. 3). He requested immediate formulation of a joint Service program to assure maximum useful software commonality in the DoD. A working group was formed from official representatives of the Military Departments and chaired by DDR&E. Representatives from OASD-I&L, OASD-Comptroller, and the Defense Communications Agency, and NASA also participated. The author acted as technical advisor.

A major step in achieving software commonality will be the adoption of a very few (possible only one*) common programming languages to be used for the design, development, support, and maintenance of all digital computer software for embedded computer applications in the DoD. Such a language would need to encompass the specialized needs of the intended DoD software applications, be able to support best current software practice, be complete and unambiguous in its definition, and be capable of supporting enforceable standards. As a short-term effort, it will have to be practically and efficiently implementable with existing software technology.

Programming languages are neither the cause of nor the solution to software problems, but because of the central role they play in all software activity, they can either aggravate existing problems or simplify their solution. Adoption of a single common language alone, will not make software more responsive to user needs, reduce software design or programming errors, make software more reliable, reduce software costs, simplify test and maintenance, increase programmer productivity, improve object efficiency, or reduce untimely delivery of software.

However, adoption of an appropriate common programming language may remove many of the barriers to solving software problems. It may lessen the communications barriers which prevent new systems from using the experiences of earlier, similar systems to full advantage. It may reduce the burden and delay of designing, building, and maintaining languages, compilers, support software, and software tools for individual projects and permit them to be concentrated on their applications. It may remove the dependence on original software vendors and increase competition. It may encourage development of better tools, both through pooling of costs within the DoD and by creating a larger market for independently developed software tools and aids.

*For convenience hereafter, we use the singular to refer to the minimum number of languages needed.

The scope of the common programming language effort has been limited to applications subsumed by embedded computer systems because there are several software problems unique to embedded computer systems, because such systems represent the majority of software costs in the DoD, because they are the major application areas in which there is no widely used language currently, because they represent the applications with the most pressing software problems, and because they are the only area in which most programming is currently done in assembly or machine languages. The diversity of functions performed by embedded computer systems, however, guarantees that the most characteristics needed in data processing and scientific programming will be included in the requirements for a embedded computer system language.

Embedded computer systems software tends to be large (involving many programmers working together), and to be long-lived (with several turnovers of software personnel during its lifetime). Run-time efficiency is important because of real-time constraints. Delayed deliveries can be extremely expensive in indirect costs from loss in the useful life of the military systems in which the software is embedded. Programming errors can have catastrophic consequences.

2. Organization and Method

The needed language characteristics will be used as qualification criteria for candidate languages. They attempt to address each major issue associated with the selection of a common language, and where there is a definitive reason, the characteristic prescribes a resolution to the issue. In other cases, they provide only guidelines or decision criteria.

The needed characteristics were developed through a 9-month-long feedback process involving the Working Group, IDA, many commands and offices within the Military Departments, and several

outside organizations. These included all potential users who could be identified. In all, over 200 individuals from 85 DoD organizations, 26 industrial contractors, 16 universities, and 7 other organizations participated.

The effort to identify the needed technical characteristics for the common DoD programming language began with a meeting of technical personnel representing the Military Departments at IDA on April 4 to 11, 1975. That meeting generated a trial set of language characteristics which was intentionally vague and inconsistent, but provided the stimulus which enabled the potential users to characterize their needs for a programming language and to point out the factors which affect their choice of language.

This trial set of characteristics was widely distributed by the Military Departments with a request that the recipients submit their own set of language requirements in response. Outside contractors, contacted by the individual offices that deal with them, responded overwhelmingly. The responses were first sent to Working Group representatives of the individual Departments for coordination within their departments and on to IDA.

IDA's task was to analyze, interpret, and resolve the responses into a consistent and unambiguous set of needed characteristics. In many cases, this involved direct consultation with individual contributors. The result was an extensive document which explained some of the implications, noted the tradeoffs which were considered, and, in general, provided the rationale behind the listed characteristics.

The whole process was then repeated. The revised document was distributed by the Services and, again, many thoughtful and helpful responses were received, processed, analysed, and reconciled by IDA. A revised version of the characteristics was then prepared. This set of requirements involved few major changes

in substance, led to a contraction in the number of needed characteristics through consolidation of related items, concentrated on clarification, and led to the elimination or weakening of requirements, which, although desirable, are not feasible with existing programming language technology. At a session held December 10-12, 1975, the set of needed technical characteristics for a common DoD programming language underwent several minor revisions based on the official coordinated inputs of each Military Department and a detailed review by the Working Group and representatives of several interested organizations within the Services. Further changes are not anticipated. We hope the current set is neither vague nor unnecessarily limiting; it represents a few compromises, but appears to be technically sound and achievable with existing technology and is a consensus of the Military Departments which individually approved it early in 1976.

The resulting characteristics, presented in Chapter V and VI of this report, are discursive rather than quantitative because there are few useful quantitative measures of software or of programming languages. The depth of discussion varies according to the characteristic. The relative merit of alternative approaches, the trade-offs involved, and the rationale for the final choice are given in greatest detail for those language characteristics which have greatest impact on the language selection, have several competing approaches, or were resolved in apparent conflict with conventional wisdom.

C. Findings

The Higher Order Programming Language Working Group identified 78 needed characteristics. Major characteristics, listed below, were abstracted from that list. There is no significance in the order of presentation.

1. The common programming language can only achieve its breadth of application and flexibility of expression by having a few, general, abstract concepts and structures which can be applied in many combinations. It should not be a conglomerate of many special features of limited application or of features with many special cases in their abstract definitions.

2. The common language should have a high degree of generality and flexibility at compile time, but should be static at run time. The language itself should not require dynamic storage allocation or the presence of an operating system in its object machine.

3. The language should require its users to specify the type of data and operations, the range and precision of numeric data, and the action to be taken under each alternative condition in its programs. These all represent information that is known to the programmer and needed by those who must maintain software. These kinds of information can also be helpful to the translator in producing more optimal code and can aid in testing and debugging programs.

4. The language should require redundant (not duplicate) specifications in programs so that many program errors can be detected automatically. For example, both formal and actual parameters should be (possibly implicitly) specialized by type to permit compatibility checking. A combination of typed data and type independent precedence levels of operators will ensure that the structure of expressions can be verified both syntactically and semantically.

5. The language should permit definition of new data types and operations, thus allowing specialization to particular applications without modification of the language definition, its translator, or its support software. Type definitions may also enable its use in unforeseen applications.

6. The language should permit its users to distinguish between the abstract and concrete representation of data, between the functional and algorithmic representation of operations, and between the scope of allocation and the scope of access for variables. The ability to separate specifications of these kinds means that the logical structure and intent of programs need not be obscured by those aspects which are concerned only with adherence to physical constraints of the underlying machine.

7. The language itself should not be optimized to any particular criterion, such as object code speed, object code size, ease of program modification, program clarity, or ease of programming, but should provide facilities which permit individual programs to be optimized to any of these criteria. Optimization criteria are often application- or task-dependent.

8. The language should provide special facilities to simplify the description and implementation of programs with real-time constraints and real-time interaction with multiple peripheral devices.

9. The language should have a complete and unambiguous definition and should not be dependent on any particular object machine or operating system structure.

10. The common language should be composed of existing language features, but may not be exactly any existing language well known to most potential users. Thus far, no combination of the needed characteristics which is not achievable with existing programming language technology has been found and, if any were, they would be interpreted as cause for reducing the needed characteristics. On the other hand, since no identified language satisfies all the needed characteristics, some modification of existing languages will be necessary. Furthermore, regardless of the language selected, the diversity of languages currently used guarantees that it will be new to most of its potential users. The characteristics dictate a language which draws its features in obvious ways from existing languages and which avoids many specific recognized deficiencies of currently used languages.

11. A major emphasis should be on the support supplied with the language. Ultimately, the success of the common language effort will depend on the acceptance of the language by DoD software developers and, to a large extent, that will depend on the availability and accessibility of supported compilers, software aids, and libraries for the language.

From preliminary analysis, the identified needed technical characteristics for the Common Higher Order Programming Language for military applications appear to be self-consistent, to conform to the established language design/selection criteria, to be acceptable to the Military Departments, and to be achievable with existing software and programming language technology. More analysis is required, however, particularly on the feasibility of achieving all the technical requirements simultaneously.

Detailed examination of existing language features, language design techniques, and compiler implementation and optimization methods are needed.

The process of identifying the needed technical characteristics for a Common Higher Order programming language also uncovered several possible properties of the programming environment, translators, and management of the language which the Working Group thought will be important to the success of the common-language effort. These properties include the availability of language-associated software development tools, standard libraries, translator options, and source language diagnostics. They prohibit superset and subset implementations, recommended multiple-object-machine translators, and require self-implementation of the language. Most importantly, they require user documentation, configuration management, standards, control, and support for both the language and its libraries.

I. INTRODUCTION

This paper is concerned with criteria and issues that will have an impact on the needed technical characteristics for a Common Higher Order Programming Language for military applications.

Chapter I gives an introduction to the software and programming language problems in DoD, the purpose of the common language effort, and some related issues.

Chapter II presents some conflicts that arise in any language design or selection process and describes their resolution for the common-language effort.

Chapter III reviews some of the major problems affecting software design, development, maintenance, and use in the DoD.

Chapter IV presents the language design criteria which helped determine the needed characteristics. These criteria fall into three major categories: those which satisfy specialized application requirements, those which address recognized existing software problems, and those which are intended to assure that the resulting language can serve as a common language.

Chapter V gives the needed technical characteristics for the common language, while Chapter VI provides additional requirements related to the programming environment of the resulting language, to its translators, and to a number of management issues concerning its definition, standards, support, and control. Many of these issues will have direct or indirect effects on the technical acceptability of candidate languages.

A. THE PROBLEM

> *As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them--it has created the problem of using its products.*
>
> *E. W. Dijkstra in 1972 Turing Award Lecture*

The past 25 years of digital computer hardware history are characterized by orders-of-magnitude increases in computing speed, memory capacity, and reliability. At the same time, the physical size, power consumption, and cost of computer hardware have decreased by several orders of magnitude. These trends have led to inflated expectations and expanded use of digital computers not only to automate tasks that previously had been performed manually, but tasks not seriously considered heretofore.

The burden of increased expectations for computer systems has fallen on software. Software is the collection of computer programs which give direction to the computer hardware, tailor the computer to serve the needs of the application, and specify the sequencing of individual actions to be taken by the computer under prescribed conditions. Demands on the design, development, and maintenance of computer software are magnified by increases in the speed, capacity, and reliability of computer hardware.

1. Software Costs

Although little reliable information on the costs of software in the Department of Defense (DoD) is available in a clearly identifiable form, some reasonable estimates have been reported (Ref. 2). Total annual expenditures for system analysis, design and programming of software in DoD are estimated

at $3 to $3.5 billion, divided among the Military Departments as follows: Army 23 percent, Navy 36 percent, Air Force 36 percent, and other DoD agencies 5 percent. Another study (Ref. 1) has provided some estimates of the software costs by application, as shown below. If management and logistic information systems are taken as primarily data processing, and if aircraft and missile engineering and production are taken as primarily scientific programming, then the remainder, called embedded computer systems, constitutes 55 to 75 percent of the total software cost.

APPLICATION	PERCENT
Research, Development, Test, and Evaluation	28
Intelligence and Communication, Command and Control	19
Avionics	9
Aircraft and Missile Engineering and Production	5
Management Information Systems	14
Logistic Information Systems	5
Other and Indirect Costs	20
	100

The process of design, implementation, test, documentation, and maintenance of software can generally be called programming. Programming activity is constrained by the availability of dollars, real time, machine resources, competent programming personnel, and programming tools. As with any activity in which expectations exceed the available capability, something must give. In this case, the symptons appear in the form of software which is nonresponsive to user needs, unreliable, inflexible, difficult to maintain, and not reusable. The solution to the software problem will be complex, involving more and better requirements validation, software design techniques, design analysis, management visibility, discipline in software

development, program validation and testing methods, maintenance documentation, education of programming personnel, and software tools. Because there are so many aspects to the software problem and its solution, improvements in one area are often difficult to measure and have only indirect impact on the total problem.

2. Programming Language

The programming language is the one software tool which pervades all software activity from design and development through maintenance. It is a formal notational mechanism with which the programmer specifies desired computation. The programming language provides the set of software building blocks in the form of variables, data structures, operations, and control structures. With it, the programmer can

- Design, build, and refine his programs.
- Obtain the feedback enabling him to test, verify, and debug his programs.
- Assemble and manage the component parts of a software system.

Together with its programs, the programming language provides the only complete and accurate documentation of the software. It is, itself, a computer program in the form of a translator converting programs of the language into strings of symbols that can be directly interpreted by some object machine. It defines an abstract machine which associates an interpretation with each program of the language, independent of any hardware, and it is a language for communicating procedures, techniques, and algorithms among software personnel.

3. Lack of Commonality

There are a number of widely held perceptions about the ill effects of the lack of programming language commonality in the DoD. Although these can be substantiated only by examples,

and their true extent is unknown, they have provided much of the incentive and generated much of the initiative for the common-language effort. The lack of programming language commonality in DoD embedded computer applications may:

- Require duplication in training and maintenance for the language, its translators, the associated software support packages, and all the common functions needed to use any language effectively. Programming languages are themselves implemented as computer programs which must be designed, developed, and maintained. The cost and effort required for implementation, maintenance, and training increases with the number of languages in active use.
- Minimize communications among software practitioners and retards technology transfer. The strengths and weaknesses of the programming language affect the way one organizes programs, the techniques employed, and the approaches used in solving computational problems. A programming language provides much of the technical vocabulary needed to communicate about programs and problem-solving methods in software. Consequently, diversity in languages establishes artificial boundaries among software practitioners, complicates communication, reduces understanding and cooperation, and may lead to distrust and mutual criticism. A prime example is COBOL. COBOL is a thoughtfully designed language well suited to data processing applications, where it is used almost exclusively. Its effectiveness is demonstrated by the stability of its design for about 15 years. Yet, those who have not used COBOL are almost universally critical of the language. They know little about the language, except that it is different, they find its appearance aesthetically displeasing, and are sure they would not like it.

- Result in support software being project-unique and tie software maintenance to the original developer. Programming languages are often developed to support individual projects in DoD. Typically, the language will be developed as part of the project effort and used only for that project. Although the language is developed at government expense, the original software vendor is both its developer and only user. This tends to tie maintenance of the application software to that vendor. Thus, a language can be seen as a handy device to assure a continued flow of business to the contractor over the life of a system. This tendency is strengthened in the usual situation in which the translator and support tools for the language are written in another language which is proprietary to the vendor.
- Diffuse expenditures for support and maintenance software so only the most primitive software aids are developed, but repeatedly. Software tools and aids in the form of compilers, interpreters, diagnostic aids, debugging packages, code optimizers, automatic testing systems, program editing systems, and many more are programming-language-dependent, and, consequently, must be developed for each new language. Projects are, of necessity, application oriented; their primary goal must be to develop the application software. Project personnel have neither the inclination, time, funds, nor expertise to develop more powerful or more generally useful software tools.
- Limit the applicability of new support software. Even if a variety of useful tools were developed for some language, the benefits would be limited to the users of that language. Ideally, generally useful software tools should be independently developed and maintained and made available to any project, but the diversity of language guarantees that any such independent development will have only limited payoff.

- Create a situation in which the adoption of an existing language by a new project is often more risky and less cost-effective (at least during development) than developing a new specialized language. There must always be a trade-off between a specialized language tailored to the application and a more general language whose support and maintenance costs can be shared across many projects. As long as there is no common, widely used programming language for embedded computer applications which has useful, independently developed and maintained off-the-shelf support tools, there is little advantage to selecting an existing language for a new project. Developing a new language will not be significantly more expensive than developing a new compiler for an existing language and may avoid unnecessary generality while providing features especially well-suited to the application.

4. Common Language

The intent of the common language effort is to identify a language for DoD which will eventually supplant all languages in military applications for which there is currently no common language. These include weapon systems, command and control, test equipment, communications, avionics, training, systems programming, and embedded computer system support software. There is no intent to supplant COBOL for data processing applications or FORTRAN for scientific applications. Because weapon systems and command and control applications include both data processing and numeric processing functions, however, the resulting common language should be suited to those applications.

In several ways, COBOL is a model for the common language effort. It may not be a viable candidate as a common language for embedded computer systems because it represents the software technology and programming practice of 15 years ago, it

was designed specifically for data processing, and it lacks many of the special capabilities needed in embedded computer systems software. Nevertheless, the adoption of standards early in its development, the consistency of design throughout the language, the early involvement and support by industry, the uniformity of its implementations, the stability of its design, and its concern for potential users are all characteristics worth emulating.

Another successful example of language commonality is CORAL-66. In 1970, the United Kingdom Ministry of Defence formally adopted CORAL-66 as the standard programming language for real-time systems. The official policy disseminated to industry included a requirement that all computers used in weapon systems must have a tested and approved CORAL-66 compiler. The result has been not only language commonality within the United Kingdom military establishment, but wide acceptance in the commercial sector as well.

5. Morse Code Experiment

There are few useful measures of quality, performance, or cost. There is insufficient data to determine quantitatively the current situation in software, let alone predict the effects of greater software commonality. A recent experiment, however, contributes to the optimism for better software when existing software tools are more widely accessible.

There have been claims from the research community that the combination of powerful and stable software tools (including language), proper methodology, and competent personnel can improve the cost and performance of software by orders of magnitude for large complex problems. The Defense Advanced Projects Research Agency (DARPA) recently funded an experiment to test some of these claims. Researchers at MIT were asked to build a software system to solve a real, nontrivial, ill-defined problem in an application with which they were unfamiliar but

using tools and methodologies they had developed earlier under DARPA sponsorship.

The problem was to implement a system that could recognize Morse code generated by a human operator in the presence of transmission noise. The product was impressive in its ability to recognize Morse code, but was able only to operate at one-half to one-third real time. When this deficiency was pointed out, the MIT researchers undertook a two-week effort which improved response by a factor of 30 to 50. The major claim made of their approach is the ease of making changes, whether for correctness, to meet new requirements, or to improve performance.

A rule of thumb used in software development says that a programmer will produce an average of ten debugged instructions per day. The Morse code project took a grand total of 54 man-months. The final system consisted of object programs, object program tables, and run-time support software which was developed independently of the project. There were also a number of object programs developed and later discarded. Some Morse code support software was developed to help implement the system but was not a part of the final system. Depending on which subsystems are included in the instruction count, the instructions per man-day range from 139 to 909. The results are detailed in Table 1. The figure of 625 instructions per man-day is most consistent with usual practice and the figure of 491 is probably the most fair. The RAND CCIP-85 Study (Ref. 1) pointed out that an increase of from 10 to 11 instructions per day would, in theory, save the Air Force $100 million per year.

Table 1. Results of Morse Code Experiment

	NO. OF INSTRUCTIONS		INSTRUCTIONS PER MAN-DAY	
	Excluding Program Tables	Including Tables	Excluding Tables	Including Tables
Object Program Only	158,000	478,000	139	422
Total Codes Written	559,000	879,000	493	775
Total Codes in Final System or its Support	389,000	709,000	343	625
Total Codes Written and Not Discarded	237,000	557,000	209	491
Total Codes in Final System	210,000	530,000	185	467
Total Codes Written, in Final System, or in its Support	711,000	1,031,000	627	909

B. PURPOSE OF THE COMMON PROGRAMMING LANGUAGE EFFORT

> *And the Lord said, Behold, the people is one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do.*
>
> *--Genesis XI 6*

The purpose of the Common Programming Language effort is to achieve maximum useful software commonality in DoD embedded computer applications through a reduction in the number of programming languages used.

Reducing the number of programming languages may be a slow and tedious process, since languages used in existing systems can be phased out only as the systems become obsolete or go through major upgrades. Incentives in the form of supported, easily accessible, and easily used software tools and aids are needed for the languages which remain. The standards and stability in the remaining languages should not impede the use of new software tools and methods, hinder the adoption of new programming techniques, or stifle innovation. When new languages are introduced, and they must be to take advantage of new software and programming language technology, it should be done in a controlled manner and only when there is expectation of major benefits and full understanding of the trade-offs.

Software commonality refers to the reuse of computer programs, software subsystems, or methodologies, either directly or after minor modification. The value of software commonality derives not only from reduction in redundant software development, but from lower costs for training and maintenance of the resulting systems, more timely system development, lower risk in software design and development, and better communication among software practitioners.

The benefits of reusable software are greatest for support software, including compilers, verifiers, programming and debugging systems, and optimizers. Support tools are often ineffective, because they are reinvented and rebuilt for each new project. There is seldom the time or money to perfect the support tools or to provide any but the most primitive capabilities. If the same software is widely used, costs can be shared over many projects and each effort can build on its predecessors.

1. A Common Programming Language

Software commonality can be achieved only through the adoption of a common programming language. The programming language is central to the development of software. The software tools and aids are built around specific programming languages. The compilers and other software tools are themselves the most widely used computer programs and, as such, can especially profit from the attendant improvements in reusability, training, maintenance, timeliness, risk, and communication.

The fewer the programming languages the greater the leverage associated with those that remain. The common-language effort has as a goal minimization of the number of programming languages in DoD. The common language is intended to be a single or minimal set, of general-purpose programming language that will eventually replace the many hundreds of general-purpose languages being used currently in the DoD. The assumption that a single general-purpose language will suffice must remain until specific needs or conflicting language requirements demonstrate a need for more than one. Neither should general-purpose languages be confused with application packages, which are sometimes called, "application-oriented languages". Unlike general-purpose languages, application-oriented languages can be used to describe computations only in limited application areas, are designed for use by practitioners in the application and not

by computer programmers, are usually interactive, and are often nonprocedural. The adoption of a common programming language should lead to the implementation and support for standard application packages. Finally, although a single, general-purpose language is desired, there is no intent to impose another language where useful language commonality already exists (*e.g.*, among COBOL users and in scientific uses of FORTRAN). Neither is it feasible to rewrite existing programs, regardless of the merits of a standard language.

Although adoption of a common programming language is necessary to obtain the benefits of software commonality, it is not sufficient. There have been many past efforts which merely attempted to standardize on a language's syntax and semantics while ignoring performance properties and program development aids. There must be commonality among language-related support tools, commonality in compiler performance, and support and maintenance for the language, for its software development and maintenance aids, and for its library of application routines.

2. High-Order vs. Low-Level Programming Language

The distinction between high-order and low-level languages is similar to that between people and machines. Any programming language should present an analog to the underlying machine in a form more amenable to human use. Low-level languages are machine-oriented and simplify the translation process at the expense of human resources. The higher the level of the language the more it caters to the needs of the programmer, the greater the portion of software development which is automated, and the less visible are the underlying machine resources. Programming here, of course, encompasses not just coding, but the entire spectrum of software design, development, and maintenance.

In commenting on section V.J. of this report, E. W. Dijkstra said:

> *I can enlarge on that: in the past, when we used 'low-level language' it was considered to be the purpose of our programs to instruct our machines; now, when using 'high-order language', we would like to regard it as the purpose of our machines to execute our programs. Run time efficiency can be viewed as a mismatch between the program as stated and the machinery executing it. The difference between past and present is that, in the past, the programmer was always blamed for such a mismatch: he should have written a more efficient, more 'cunning' program! With the programming discipline acquiring some maturity, with a better understanding of what it means to write a program so that the belief in its correctness can be justified, we tend to accept such a program as 'a good program' if matching hardware is thinkable, and if, with respect to a given machine, the aforementioned mismatch then occurs, we now tend to blame that computer as ill-designed, inadequate, and unsuitable for proper usage. In such a situation, there are only a few ways out of the dilemma: (1) accept the mismatch, (2) continue bit pushing in the old way, with all the known ill-effects, and (3) reject the hardware, because it has been identified as inadequate.*

A higher-order language permits automation of the more repetitive aspects of software development in return for greater constraints on the programmer. The high-level language programmer is deprived of dangerous capabilities, such as being able to create self-modifying programs, to do arithmetic on machine addresses, and to use fixed-point operations on floating-point numbers. In return, the programmer is able to partition his program into logically meaningful parts, is guaranteed that updating one variable will not affect others, and is given warning when he violates his own assumptions and stated conditions.

The costs in machine resources which must be paid for greater automation of software development through the use of high-order programming languages can be paid at compile time (*i.e.*, the time of translation) or at run time (*i.e.*, the time the software is used). Many of the widely used HOLs simplify the programming task by providing general-purpose mechanisms and many automatic defaults so that the programmer not only has the advantage of intuitively meaningful structures and notations, but is relieved of having to specify his intentions, assumptions, and the detailed constraints on his programming problem. Consequently, information available to the programmer is hidden from the compiler and maintenance personnel and must be derived dynamically from the program at run time, thus imposing much greater run time costs than would be associated with a corresponding program written in machine language.

There are several ways out of this dilemma. For programs which are to be executed only a few times or for other reasons have unimportant run time costs, the solution has been to admit to the greater run time costs, incorporate the run time environment into the translator to form an interpreter, and take full advantage of the machine-independent high-order language. Where run time costs are important, at least four approaches have been tried. The most widely used approach in DoD has been to allow portions of the HOL program to be written in machine language. This permits the programmer to optimize his programs to any degree within his capability but defeats the purpose of the HOL. Another approach used in DoD, but most popular in the commercial and scientific world, has been the optimizing compiler. Large, sophisticated compilers have been built to rework the object code to produce an optimal run time program. Although these systems impose considerable compile time costs, they have seldom been able to produce codes comparable to those of good machine language programmers. The third approach admits that a compiler-produced code is not as efficient as handwritten

code for small static programs, but points out that programs in which object code efficiency is important tend to be large programs which are modified many times. The mass of detail which must be processed for effective optimization of a large software package may be too much for even the best programmer, and when success is achieved, it may be very transitory because assumptions under which the optimizations were made change as system requirements change.

The fourth approach, and one which seems most reasonable for the Common HOL effort, emphasizes software reliability, program maintainability, and run time efficiency in the context of the current programming language model (*e.g.*, ALGOL, JOVIAL, PL/1 and the like). It gives up some programming ease by using a language which requires the programmer to make his assumptions and intentions explicit in his programs and which prevents hiding information from the compiler and those who maintain the programs. Greater software reliability results because more information is available to compiler for compile time error detection, software is more easily maintained and modified because it is more readable and comprehensible, and execution is more efficient because more information is available to the compiler for optimization and because more decisions are bound at compile time. High-level language programs should contain a great deal of information of value to the compiler as well as to those who must maintain the program. This in no way conflicts with the characterization of the HOL as being oriented toward the programmer, human problem solving, and particular application areas at the exclusion of machine-dependent characteristics. Programming language features which aid the compiler in the generation of efficient object code should have a form and meaning which will contribute to the understandability of the program as well, should be translator independent, and, to the degree possible, object machine independent.

C. OTHER ISSUES

There are a number of important issues and potential problems associated with the Common DoD Programming Language effort and with the discovery and use of the needed technical characteristics for such a language. This section discusses some of these issues and gives the resolution where there has been a decision by the working group.

1. Scope

The Common Higher Order Programming Language effort has been limited to embedded computer systems applications because (1) the majority of software costs in the DoD are associated with weapons systems applications, (2) COBOL and FORTRAN already satisfy many of the commonality goals of this effort for data processing and scientific applications, respectively, the two major areas which have not been included, and (3) the large number of unique, nonstandard, general-purpose programming languages used in the DoD are used in embedded systems applications.

The scope of the effort has not been further restricted within embedded systems applications because (1) specialized applications within weapons systems have similar software problems, (2) embedded systems applications are not pure and require computations in many specialized areas within the same system, (3) the technical requirements for the individual applications have proven to be nearly identical, and (4) no conflicting requirements have been found.

2. Application-Oriented Languages

The Common Higher Order Programming Language is intended to eventually supplant all general-purpose programming language used in embedded systems applications in the DoD. It is not intended to replace application-oriented languages. Application-oriented languages are similar to programming languages in that they enable their user to describe a computation which will be carried

out by a digital computer. They are unlike general-purpose languages in that they provide very specialized capabilities for a restricted problem domain, they are intended for use by those familiar with the application and usually do not require specific programming knowledge, they are often nonprocedural, and in many cases are accessible interactively. Any application package is an example of an application-specific language. The Common Higher Order Programming Language effort is concerned with the general-purpose procedural programming languages used to implement applications and systems software, and is not intended to replace application-oriented software.

3. Effect on Software Expenditures

The projected overall benefits of any standardization should exceed its disadvantages. Ideally, there should be a complete cost-benefit analysis, comparing costs with and without standardization, including life-cycle costs. This is not feasible for software/programming languages, because their costs are diffused throughout weapons systems procurements and are seldom identifiable. We do know, however, that most costs are for personnel, that there are hundreds of general-purpose languages in use in DoD, that much software work is duplicative because similar software (particularly system and support software) must be redundantly developed for each language, and that the diversity of programming language has complicated the development of widely applicable programming tools and aids which could alleviate or reduce many of the recognized software problems.

It is likely that adoption of a Common Programming Language will result in better communication among software practitioners; easier transfer of new software technology to production systems; greater software reusability; easier transfer of personnel among projects; greater visibility of underlying software problems; increased programmer productivity; improved

software quality; and development of better and more applicable software design, development, and maintenance aids. Reduced costs might be expected, not just from the adoption of a common language, but from the prohibition on the development of other new programming languages. Development costs for other new programming languages will be eliminated (prior to January 1975, there were typically several at any given time under development by elements within each Military Department). Compiler costs will be reduced, even when new digital computers are introduced, because a common language with its machine-independent portions written in its own language means that a new computer can be made accessible by reimplementing only the code-generation portion. Because tools, programming aids, and other support software will be more widely applicable, the total cost of its development and maintenance should be reduced. Similarly, the training costs for a single widely used language should be less than those for the many project-unique languages. Finally, as with any successful standardization effort, the common language should encourage competition in software development and give more freedom to change vendors.

It does not follow, however, that total software expenditures in the DoD will be reduced. Benefits of a successful common-language effort must be limited to new software developments. A primary impediment to reliable software is change to existing computer programs. A common language might be used in new software efforts, but it is seldom economical to reimplement existing systems. More importantly, constantly increasing personnel costs, more demanding military system requirements, and continuing budget pressures have led to more and more automation. Computer software is a major component of electronic equipment procurements, so any increase in software productivity or quality will likely accelerate this trend. Success in the common-language effort may reduce the cost of software and increase its quality, but these will likely be accompanied by increases in software expenditures.

4. Effect on Software and Programming Language R&D

The adoption of a common language should give greater visibility to software, should separate the language design issues from the more important software problems, and should provide a base for comparing software techniques. It should provide a community of users who can share the design, development, and maintenance costs of more capable software tools. It should provide a vehicle for transfer of software technology from research and development to practical use. It should provide a bigger market for individual software tools (which are often specific-language oriented) and should, therefore, amplify any benefits of the National Software Works. The separation of language issues from other problems of software development may serve to identify language deficiencies, problems, and needs for innovation in language design, and thus lead to increased programming language research and development.

All these effects tend to give greater visibility to the real underlying software problems and to the importance and benefits of their solution. A common language may give visibility to the sparseness of current software research and development efforts, and point out the need for improved software development methods, techniques, tools, and aids. It will likely lead to an expanded DoD R&D program in software and in programming languages, but one directed more toward finding practical solutions to important recognized problems.

5. Direct Costs of Common-Language Effort

There will also be several costs associated with obtaining a common language. There are the development costs for the language itself; there are design, development, implementation, maintenance costs for its compilers, support software, and related programming aids; and there are training costs for its users. In each of these cost areas, however, the adoption of a common language should result in reduced expenditures, because

the common language effort replaces many similar local efforts that would otherwise have taken place within the Military Departments. Even for a single large military system development, independent development of the programming language, compilers, and software support tools will remove those efforts from the application software development and thereby reduce the development time for the application software (timeliness is a major indirect cost of software). That one development can be used by many projects, of course, eliminates many redundant expenditures. Finally, without speculating on the total cost of the common language effort, it should be noted that at a level of $3 million per year, it would be less than 0.001 x the annual DoD software costs and at $10 million (*i.e.*, approximately 100 man-years per year) it would be much less than 1 percent of software costs.

6. Standardization

Standards programs should not be undertaken unless certain criteria are met. There should be several potential users of the standard (in this case all new embedded computer application in DoD). There should be a mature technology well in hand (in this case the FORTRAN, COBOL, ALGOL, PL 1-like programming language technology). There should be a potential market large enough to support at least one contractor for several years. The projected overall benefits of standardization should exceed its disadvantages (see previous subsection). And, adoption of the standard language by individual systems should not be a major problem (in this case, it should be no worse than adopting a nonstandard language, and much easier, providing the common language is widely used and well-supported).

7. A New Language

It is most desirable that the selected common language be an existing language, and if that is not feasible, that it be a modification of an existing language. Given the identified requirements, it is likely that most features of the selected language will be familiar to most DoD users, and that it will not be exactly compatible with any existing language implementation in the DoD. The familiar features are likely because the requirements dictate a FORTRAN, ALGOL, PL 1-like language and were selected to be compatible with existing programming language technology. The incompatibilities of existing implementations guarantee that it will differ from almost all implementations of languages used currently in the DoD. The selected language will be new to almost all its users because:

- Definitions of existing languages are vague, incomplete, and ambiguous, resulting in creation of a new incompatible dialect with each implementation.
- The selected language is to emphasize program reliability and maintainability over programming ease, the traditional goal.
- The requirements encompass the needs of all embedded computer systems applications and not just those perceived by programmers on a particular project.
- The requirements legislate away many of the known deficiencies (*e.g.*, error-prone features) of existing languages.
- The language will incorporate many of the special characteristics needed by embedded military system applications. These have been largely ignored in the more commonly used languages (which were intended primarily for scientific and data processing computations).

- Any attempt to standardize on one precise definition of any existing language cannot work, because its divergent dialects are defined by their implementations and are therefore machine-dependent.

A new language name is desirable. Even if the language is a precise definition of some umbrella name language widely used in DoD, it should be relabeled to distinguish it from the existing divergent dialects. The common language should closely resemble (particularly in semantics) many of the existing DoD languages, but which particular one is modified to obtain the selected common language is of little significance.

8. Size

Each of the characteristics described in Chapter V addresses one or several related issues in the design of a programming language. In several cases, the issues are complex and the discussion quite involved. This does not, however, imply that the selected language must be large or complex. Each needed characteristic specifies how the design/selection process will resolve some issue affecting the design, implementation, or use of the language. Where possible, they avoid choices of particular language features. Each issue must ultimately be resolved; Chapter V provides some of the analysis and rationale where there is reason for a particular resolution and provides guidelines where no clear resolution is indicated by the application requirements, relevant trade-offs, or the goals of the common language effort. The number of issues is, of course, almost independent of the resulting language and has little, if any, relation to the number of features or the size of the language.

9. Priorities

There is no ordering of priorities among the needed characteristics, because (1) the priorities are typically application dependent and, therefore, dissimilar for the various potential users in the DoD, (2) priorities are of no value whatsoever, as long as none of the characteristics are in conflict and can be achieved simultaneously, and (3) the establishment of priorities may unnecessarily serve, in effect, to eliminate the lower-priority requirements. Priorities should be considered only if and when compromise becomes necessary.

10. Consistency

It is very important that the needed characteristics be achievable in combination with low-risk technology. Examples of existing programming languages which satisfy each individual needed characteristic are known, but whether all can be satisfied together remains a question of judgment, and there are differing opinions. Any formal demonstration that they are self-consistent is probably still beyond the capability of computer science. A more pragmatic demonstration is required. Ultimately, the only acceptable proof will be one or several programming languages that satisfy the requirements. If there are conflicts, they will become apparent in the design/modification process and must be resolved at that point.

11. Committee Design

A set of needed language characteristics has been established. The modification of an existing language design requires sound engineering and design practice by qualified people and is inappropriate to the compromise process of committees. Consequently, the language will be selected from candidates that have been reviewed by persons knowledgeable in the intended applications, in the construction of compilers, and in the design of languages. The Working Group will not design or modify languages.

12. Nontechnical Needs

The success of the Common Higher Order Programming Language effort ultimately will depend not so much on the technical characteristics of the language selected as on philosophical, management support, and procurement issues. Some general approaches to these issues have been determined by the Working Group and are reported in Chapter VI.

II. MAJOR CONFLICTS IN CRITERIA AND NEEDED CHARACTERISTICS

Five major conflicts were identified in attempting to find a consistent and appropriate set of criteria. In several cases, a closer examination of what was actually intended revealed that seeming conflicts, in fact, did not exist.

A. SIMPLICITY VS. SPECIALIZATION

The common programming language must be useful for many seemingly diverse applications, each with its own specialized needs. Suitability of the language for each of the applications is essential if it is to have wide applicability. This suggests a need for a large conglomerate language with many specialized subsets. At the same time, the single most prevalent sympton of the software problem is the complexity of programs and its adverse effects on the timeliness, reliability, responsiveness, flexibility, and maintainability of software. Probably the greatest contributor to unnecessary complexity in programs is the use of overly elaborate languages with large numbers of complex features specialized in the hope of providing every anticipated application with capabilities unique to that application. The result, in many cases, is a grotesque language, expensive for everyone, understandable to none, and well-suited to few real problems.

The problem is how to satisfy simultaneously the need for simplicity and specialization in the same programming language. The only method of which we are aware is to achieve simplicity through the use of a simple, general-purpose language which has all the power necessary for all the intended applications, but has not yet specialized that power for any particular

application. Such a language would have a few general-purpose data structures, operations and control structures, each providing a single, well-defined capability, and all composable to form more specialized capabilities needed in particular applications. The language should provide a simple, consistent, and easily learned semantic and syntactic framework. There should be definition facilities within the language to permit definition of new data and operations, but only within the built-in framework, so basic understanding of programs written in the language would not be undermined by new definitions within the language. Such a language alone, however, can only provide the simplicity and the power to build data and operations for specialized applications; it alone will not make useful definitions available to the software practitioners with the applications. To be useful, and to satisfy the specialized needs of the various applications, there must be a predefined, application-oriented library of definitions available with the language. These application packages must have the same support, standardization, and control afforded the base language. As definitions, they will not, however, add to the complexity of other applications, need not affect the implementation, and will be totally independent and unable to interfere with other application subsets.

Neither should we think that simplicity and uniformity or even power in language will make programming easy. Intrinsic complexities which follow from the task will remain. The purpose of a high-order language is to remove the unnecessary complexities which arise from weaknesses in the programming language, operating system, or underlying computer hardware.

B. PROGRAMMING EASE VS. SAFETY FROM PROGRAMMING ERRORS

There is a clear trade-off between programming ease and safety. The more tolerant the programming language and the less it requires in specifications of the intent and assumptions of the programmer, the easier the coding task. A language which does not require declaration of variables, permits any type of structure of data to be used anywhere without specification, allows short cryptic identifiers, has large numbers of default conventions and coercion rules to permit the use of any operator with any operand, and is capable of assigning meaning to most strings of characters presented as a program, will be very easy to use, but also very easy to abuse. Safety from errors is enhanced by redundant specifications, by including not only what the program is to do, but what the author's intentions and assumptions are. If everything is made explicit in programs with the language providing few defaults and implicit data conversions, then translators can automatically detect not only syntax errors but a wide variety of semantic and logic errors. Considering that coding is less than one-sixth the total programming effort, and that there are major software reliability and maintenance problems, this trade-off should be resolved in favor of error avoidance and against programming ease.

Resolving this trade-off in favor of safety at the programming language level is important not only for large, long-lived weapons systems, but for any large long-lived computer program, specifically support software, interactive application packages, and software development and maintenance aids. In specialized application software, the ease with which the user (an application specialist rather than a computer specialist) can interface with the application software is of primary importance, and user requests are often small and short-lived. The application package itself, however, will often be large

and long-lived, and thus should be written and maintained in a language which favors program correctness and maintainability, even if this costs in terms of programming ease. The Common Programming Language itself need not be interactive and should not affect programming ease at the expense of other more important criteria, but it should be possible within the Common Programming Language to develop and maintain interactive application packages with convenient, easy-to-use user interfaces.

C. OBJECT EFFICIENCY VS. PROGRAM CLARITY AND CORRECTNESS

Two apparently opposing views have been suggested. One, that a simple analysis of either development or life-cycle costs shows that reliability, modifiability, and maintainability are the most important factors, and, consequently, clarity and correctness of programs must be given consideration over efficiency of the object code, which only increases the cost of computer hardware (hardware relatively cheap compared to software). In fact, if programs need not work correctly they can easily be implemented with zero cost. The other view points out real problems and applications within DoD software in which the machine capability is fixed and in which object code efficiency is of utmost importance and must be given preference over other considerations.

These views are not inconsistent with regard to the effect on programming language selection. In the vast majority of cases, clarity and correctness are more important than object code efficiency and the programming language should do the utmost to aid the programmer in developing correct and understandable programs within constraints of reasonable object efficiency. In many cases, language features which improve clarity do not adversely affect efficiency. In many cases, additional information supplied to clarify a program will permit the compiler to

use optimizations not applicable in more general cases. There remain, however, special situations in which efficiency is critical. The language should not prohibit access to machine features necessary to accomplish those optimizations when the need arises. Thus, the major criteria in selecting a programming language should be clarity and correctness of programs within the constraint of allowing generation of extremely efficient object codes when necessary.

D. MACHINE INDEPENDENCE VS. MACHINE DEPENDENCE

Object machine dependencies occur in digital computer programs for at least three reasons. First, programs are limited by the choice of capability and capacity of the resources available in the object machine environment. Programs written for machines with different memory sizes, different peripheral configurations, and specialized hardware capabilities cannot be totally machine-independent. Programming languages can, however, treat these inherent limitations in a uniform way by permitting their users to describe the resources needed in their programs and the translator producing diagnostic messages when the program requirements exceed the capabilities of the object machine.

Sometimes machine dependencies occur in programs when the same capability is provided by different mechanisms in the various object machines. These machine dependencies can be avoided when a higher-order programming language is used. When evaluating an arithmetic expression, the number of machine operations, the form of the operations, and the management of registers and storage are quite different, depending on whether the object machine has a single-address, two-address, three-address, stack, or general-register architecture. Nevertheless, the programming language can eliminate these machine dependencies from source programs by providing a single abstraction of these mechanizations in the form of algebraic expressions. Similarly, real

time clocks might be provided in the object machine as a single writable countdown register which interrupts on underflow, or as a pair of registers, one of which is a read-only counter with an interrupt when the register contents are identical. Although the mechanisms are different, they both provide the ability to cause an interrupt after (or at) a specified time. A single programming language feature can make this capability available to the source language programmer without imposing a particular object representation.

A third form of machine dependency occurs in programs in which the programmer knows that certain language constructs, operations, or programming techniques are particularly efficient or costly on his intended object machine. This form of machine dependency is sometimes necessary and is unavoidable in languages which permit the description not only of what a program is to do, but how that computation is to be accomplished. If the source language definition is complete and unambiguous and the translators implement the source as defined, then this form of machine dependency will not adversely affect the ability to correctly compile and run programs on other than the intended object machine.

A machine-independent language is one in which any of its programs can be compiled and will run correctly on any object machine of the language, provided that the program does not call for greater capability or more resources than are available on the particular object machine. This means that the language must permit the programmer to avoid unnecessary machine dependencies in his programs. It should permit the user to describe the ranges, precisions, and types of data and operations needed in his programs, rather than forcing his concern to the actual word-sizes, arithmetic type, and internal representations provided in the object machine. The programming language can and should be independent of the object machine characteristics

and the compiler. At the same time, it should be possible to write machine-dependent programs as described in the first and third paragraphs above. When a program exceeds the capacity or capabilities of the intended object machine, the error should be reported by the translator. Even the ill-effects of machine language insertions and machine-dependent data representations can be minimized by requiring that they be within the body of a conditional which is dependent on the object machine configuration.

E. GENERALITY VS. SPECIFICITY

A problem which often arises in looking at more detailed programming language characteristics is the trade-off between specialized and more general features. General features can satisfy a greater variety of needs and can be specialized to meet many, possibly unforeseen conditions. Specialized capabilities are often more efficient than specialization of general capabilities and, therefore, less expensive in use. Both points are often true in practice but the latter need not be. Generality can be achieved by consolidating many diverse cases into a single general-purpose structure which treats each as a special case, or it can be achieved by identifying the primitive building blocks from which more specialized structures are built. The latter approach has several advantages in programming languages. First, because all language features ultimately must have a representation in terms of computer hardware primitives, composable general-purpose programming language primitives which have a simple representation in hardware primitives can be used to compose specialized language structures as efficiently as could be done by building them in. Secondly, general purpose language primitives which emulate single machine language capabilities, but at the user level, will obligate the user to pay only for the capabilities he needs.

The trouble with specialized capabilities built into a programming language is that they seldom are specialized in precisely the direction needed for the problem at hand. The ALGOL-60 *for* statement is extremely useful and desirable if one's loop requires a control variable, has a sequence of possible terminal conditions affecting different iterations, and needs to be able to change the terminal value of the index variable from within the loop body. Seldom are all these capabilities needed, but all must be paid for in program clarity, language complexity, and object efficiency. A programming language should strive to provide a base of simple, single-purpose, composable primitives and leave the specialization to supported application packages and to user programs. The language primitives should be machine-independent abstractions of machine primitives which have an obvious and efficient representation in most machines.

Care must be exercised to insure that language structures which are defined within a language, instead of being built in, can be implemented efficiently. If the notational mechanism used to make a definition requires over-specifications which are not necessary to the intended structure, then the compiler has no way of knowing that these additional specifications are unnecessary and it must provide for them. Although it is not currently possible to write programs in an abstract language which specifies only the essential aspects of defined structures and then to use a compiler which will find an optimal concrete representation from that description, it is possible to separate the abstract and concrete descriptions of defined features so that the idiosyncrasies and special characteristics of a particular implementation do not interfere with the clear understanding and easy use of the defined feature.

III. THE MOST PRESSING SOFTWARE PROBLEMS*

The problems mentioned below are derived from a variety of in-house and contractor studies of the software problem in DoD as well as the Service inputs to the common-language effort. It should be noted, however, that these problems are unique neither to the military nor to software.

The cost of software is high and, therefore, its problems are worth examining in more detail. Software costs in the DoD are estimated at $3 to $3.5 billion annually. Another $2 to $3 billion is consumed in the support and operation of digital computer systems. These compare with computer hardware procurement and maintenance costs estimated at $1 to $1.5 billion per year. Approximately 70 percent of all computer costs (*i.e.*, computer hardware, software, and operations) are for personnel. Essentially all software costs are for system design, analysis, and programming personnel. Of these, 75 percent represent in-house costs.

That software costs are high does not necessarily mean that they are excessive. In some cases, computers are used to automate previously manual tasks. With rapidly rising personnel costs, declining computer hardware costs, and stable or declining software costs (for given tasks), it is likely in such cases that total costs have been reduced through the use of computers. In many more cases, the use of computers provides increased capabilities for tasks in which people are too slow, inaccurate, or otherwise ill-suited. It is difficult to place dollar values on improved or increased capabilities.

*Costs reported in this section are taken from Ref. 2.

A. RESPONSIVENESS

Software is often unresponsive to user needs. The dearth of techniques for specifying requirements and the complexity of software systems create a situation in which there is minimal understanding of the intended user's real requirement by those who must design and implement the software. By the time the system is sufficiently near completion for the user to try it, most decisions are irrevocably built into the design. There is an almost universal disregard for the building of prototype systems to resolve or clarify user requirements. The result, all too often, is software that is of little value to anyone. It should be noted that the need for prototyping applies to the common language effort as it does to other software designs.

B. RELIABILITY

Software reliability resembles hardware reliability in that it is possible to measure the mean time between failures and in that failures are not always reproducible under seemingly similar circumstances. In reality, however, software faults are quite different: software does not degrade with time; all software faults are inherent in its design; once corrected, a software fault will not reoccur; and exactly the same faults will occur under the same circumstances in multiple deployment of software. Unreliable software has just two causes: incorrect programs and erroneous input data. Incorrect programs result from transcription errors, lack of understanding of the program by its authors, and the use of logically incorrect algorithms. Software faults from bad data indicate lack of robustness in the program design, and, more specifically, failure of the program to validate the input for conformity with the program's input assumptions.

Another difficulty of software reliability is that the problem is often confused with the problems of changing user

requirements and software maintainability. Thus, although a program might be correct and its correctness will not degrade with time, user needs may change so that the program does not provide a useful service. Lack of maintainability and modifiability of programs may impair the ability to repair software design errors. In some situations, the effects of a change may be so opaque that there can be no confidence that changes will not introduce as many errors as they correct.

Software reliability is particularly important in the military environment where errors can have catastrophic consequences. It should also be remembered in this regard that redundant deployment and voting will not reduce or protect against software faults.

C. FLEXIBILITY/MAINTAINABILITY

Software is inherently flexible, modifiable, and amenable to change. It is soft in the sense that it is a collection of ideas, abstractions, and information without an essential physical form. The only rationale for implementing a system with software instead of hardware is flexibility. Software is used when the task has a short lifetime and will soon be supplanted by another task requiring a different program, when the task is sufficiently complex that many changes and modifications will be required to refine it into a workable system, when there is expectation for growth in the system and continued revision of the system requirements, and when the system is sufficiently unique that the economies are in specializing a general-purpose system instead of building a hard system.

Unfortunately, software's inherent flexibility is seldom available in practice. Software is pure design, with only symbolic form. As such, any change or modification to software is a change in its design. Design changes are easy only if we are not concerned with their consequences. To make design changes

with predictable consequences, we must thoroughly understand the design, what aspects of the design will be affected by our changes, and how those piecewise effects will affect the whole. Thus, purposeful software change and modification and its design of flexibility are determined by the completeness, correctness, and understandability of its design and documentation.

D. EXCESSIVE COST

There are wide variances in software productivity, reliability, flexibility, and cost. Programmers purportedly produce an average of 10 debugged instructions per day, but the variance is at least from 1 to 100 instructions per day. Software systems are not built from existing off-the-shelf or reusable parts, but from scratch each time, using the primitives of the current programming language. Programming tools with demonstrated software productivity increases of at least two decimal orders of magnitude for large complex software systems in research environments are unavailable for the military user. In many DoD applications, assembly languages are still widely used (and some would argue, to advantage, over the available HOLs). Finally, the lack of visibility of software to management, inaccessibility of software costs, and failure to give software the same scrutiny as hardware in the development of military systems creates a situation in which there is little cost accountability.

Computer resource limitation is probably a large factor in excessive costs. It is just as easy to add functions to a system that is full as it is to augment one that has plenty of slack. One reason that promising tools are not being widely used, that assembly language use is continuing, etc., is that computer resource limitations (fixed at the time of software design) force emphasis on minimum possible code per function.

This may also account for the dearth of off-the-shelf software. Most software customers want the product they buy to be small, fast, and cheap. They ask, why add extra effort and resources to provide general capabilities that are not needed for their particular project?

E. TIMELINESS

Many software projects have gone awry for lack of calendar time. The reasons are many: estimating techniques are poorly developed; effort is often confused with progress in software development, there is sometimes the false assumption that men and months are interchangeable; uncertainty of estimates, which assures that only the most stubborn software managers will stick by pessimistic time estimates; lack of engineering discipline in software development which makes it difficult to monitor progress; and adding additional manpower when schedule slippages are recognized.

In many systems, including large military systems, indirect costs from software slippages can far exceed the direct costs of the software. Deployment of a recent Command and Control system, with an expected life of 7 years, was delayed 6 months because the software was not ready. Since the total system cost was about $1.4 billion, the 6-month loss of system capability represents a $100 million indirect cost (Ref. 1).

F. TRANSFERABILITY

Software transferability is a special case of flexibility, but one with obvious economic consequences. Transferable software can be borrowed from one project or task and adjusted or modified to suit another. For the present, a realistic goal of transferability is that it be less expensive to move software from one machine to another than to write it from scratch. The costs of transferring software cannot be eliminated, and if

object efficiency is important, cannot be done entirely automatically. Successful reuse of software has been almost exclusively confined to mathematical subroutines in FORTRAN, data processing application packages in COBOL, and a few follow-on systems which borrowed extensively from their predecessors. A key ingredient in each of these was the use of the same programming language. It may not be possible, or even desirable, to reuse the top-level structures of applications software, but there is little reason why software design, development, test, and maintenance tools and aids and other support software should not be reusable. Neither is there reason to believe that lower-level software building blocks used to compose specific tasks must be unique to that task and cannot be constructed to advantage for common use throughout that application area. There was a time when functional commonality seemed as incredible in scientific and data processing applications as it now does to some in weapons systems, command and control, communications, and avionics.

G. EFFICIENCY

In software, efficiency is usually taken to mean the time and space utilization of a running computer program. Efficiency in this form is important because in some applications there are critical paths in the software which do press the available resources to their limit. Some applications (*e.g.*, simulation) have computational requirements in excess of even the largest computers, while mobile systems (*e.g.*, avionic, shipboard, and van-mounted) often have environmental requirements limiting their capability and performance. There are situations in which object code and object data representation are critical. In any case, resources should not be wasted.

There was a time when computer hardware was the major cost component of computer systems and hardware logic speed the major

performance limitation. Today, software costs far exceed hardware costs, and in many applications, the memory, peripheral, and communications speed are the limiting performance factors. Software costs increase rapidly as the computer reaches saturation. Major savings may be realized by planning for 50 to 75 percent computer saturation, but the tendency remains to consider only hardware in the initial design and to assume that the software will adjust.

If efficiency is taken in the broader sense of optimal use of all resources to minimize total cost (either life-cycle or initial development only), it becomes clear that there are many trade-offs, and that coding tricks at the machine level seldom can make a significant contribution. Real efficiency, even as measured by execution times, results first from the use of the most efficient algorithm, independent of its implementation, and secondly from identifying and improving those small parts of programs constituting the majority of the execution costs.

IV. LANGUAGE DESIGN CRITERIA

The Common HOL effort is concerned with the selection of a programming language which is expected to be used in a variety of applications, particularly those in which there is currently no widely used language. Implicit in this effort is the expectation that a large number of programming language users will adopt programming language new to them. Any change involves costs, and can be justified only if the resulting savings exceed the total costs of the change.

Success in the Common HOL effort depends on the accessibility, utility, and applicability of the selected language for use in individual application areas, on the benefits to be derived from its use, and on the ability of the language to remain uniform and stable for an extended period. Potential users of a language will not adopt it if it fails to satisfy the special needs of their application. The help a language provides in reducing software problems determines its utility and the benefits to be derived from its use. Among the major benefits of using a common language are reduced training costs, greater personnel mobility, wider use of common tools, and access to off-the-shelf software components. These latter benefits depend primarily on the stability of the language definition, the uniformity of its implementation, and an effective program of awareness of what is on the shelf.

Selection of a good or best language to serve as a common language implies use of value judgments which can have meaning only with reference to criteria. Criteria must be established to provide a basis for measuring the suitability and appropriateness of alternative designs during the language selection

process. Criteria tend to be general, imprecise, and not subject to quantative measure, but they should be unambiguous and provide a framework, a set of guidelines, which can be used to derive more specific characteristics that are subject to measurement.

The language-design criteria below reflect the three goals of (1) satisfying the specialized application requirements, (2) resolving existing software problems, and (3) assuring that the language can become a common language.

A. CRITERIA TO SATISFY SPECIALIZED APPLICATION REQUIREMENTS

1. Flexibility in Software Design Criteria

Software requirements of each system vary, depending upon the mission. The relative importance of execution efficiency, memory utilization, program modifiability, reliability, program production time, and the many other program design criteria vary widely from application to application, and even among the components of a single system. Consequently, the optimization criteria for software programs should not be built into the programming language. Instead, the language should be sufficiently robust (at compilation time) to allow the software designer to optimize his programs according to the criteria of greatest importance to his project. The software optimization criteria should be bound at program compilation time and not at language design time.

2. Fault-Tolerant Programs

In many weapons systems and control applications, it is essential that the programming language permit the description of computations which will continue to operate in the presence of faults, whether in the computer hardware, in input data, in operator procedures, or in other software. Crucial to fault-tolerant programs is the ability of the program to specify the action to be taken for all run time exception conditions.

3. Machine-Dependent Programs

There are several hundred models of computers in use in DoD. In many applications, they have unique configurations not compatible with general-purpose installations. These computers may interface with sensors or control equipment such as a radar. There are sometimes specialized computer equipments such as associative memories, real-time clocks, analog devices, and special-function boxes to aid particular computations. Programs must have access to these machine-dependent capabilities.

4. Real-Time Capability

Some applications require that faces be between the computational solution and equipment or people in real time. The programming language used in these applications must, therefore, give access to a real-time clock, allow specification of the maximum duration for execution of designated parts of the computation, and permit the programmer to specify the action to be taken upon passage of designated time intervals. These applications include monitoring of sensors; control of equipment; display; and operator input processing in applications such as avionics, command and control, communications, and training. Real-time programs may require access of time of day and interval timers, the ability to respond at periodic intervals, to service interrupts within a limited time, and to predict computation times accurately. The time quantities which must be dealt with vary from microseconds for device interface handling, through milliseconds in sensor monitoring, seconds in control applications, to days in report generation.

5. System-Programming Capability

Many applications use dedicated computers because they cannot afford the overhead and do not require the generality of general-purpose operating systems. For example, avionics, tactical systems, communications, and process control applications

include development of specialized executive systems. System programming capability is also needed for the development and maintenance of general-purpose operating systems and other support software.

6. Data Base Handling Capability

In many applications, including command and control; data processing; training; and software design, development, and maintenance it is necessary to access, manipulate, and display large quantities of data. Much of this data is symbolic or textual rather than numeric, and must be organized in an orderly and accessible fashion. Memory space rather than execution time is often the critical resource in data handling applications; large peripheral storage devices must be employed, and programs must be able to process densely packed data.

7. Numeric Processing Capability

Numeric processing capability is essential to many applications, including simulation, sensor processing, equipment control, and general-purpose engineering and scientific applications. In some environments, only fixed-point arithmetic is available on the object computers.

B. CRITERIA ADDRESSING EXISTING SOFTWARE PROBLEMS

1. Simple Source Language

The role of unnecessary complexity as the main source of problems in the use of high-order programming languages cannot be overemphasized. Simplicity in a programming language means a small language with few special cases, each feature simple in meaning and implementation, uniform syntactic forms and consistent interpretations when several special cases must be provided. There are conglomerate languages so large, diverse, and complex that programmers are not expected to understand the whole language, but only those subsets applicable to their

problems. Partitions between subsets are often not well drawn and there is little consistency among the subsets, so that when something goes wrong in a program it may invoke language features totally foreign to the authors' understanding. Even if the system detects the error, the diagnostic will not be meaningful to the programmer. *Ad hoc* language designs which have attempted to satisfy every application by providing specialized features for each special problem result in languages that are difficult to learn, impossible to implement consistently, and which guarantee unreadable, inflexible, and nontransferable software.

Untimely delivery of software is primarily the result of an inability to integrate the separate components of a large software package. The integration problem is a direct result of software interfaces too complex and ill-defined to be fully understood in the same way by all parties using them. Lack of software flexibility and maintainability is the unavoidable consequence of programs and programming languages so complex that no one can predict the consequences of program changes. Language complexity contributes to the nontransferability by ensuring that few installations will be able to afford implementation of the full language and that no two installations will implement features with exactly the same semantics. The result is implementation-dependent programs incomprehensible and unusable anywhere but where written. Software productivity depends on the ability to reuse existing software, on design, coding, and maintenance efficiency, and on the usability of the software design, development, and maintenance tools. The only hope of significantly improving software productivity is the ability to reuse software, particularly support software and software tools. This cannot happen as long as programs are incomprehensible, unpredictable, and unmodifiable. Finally, efficient programs cannot be written in languages that employ highly specialized complex features which do not themselves have efficient representations in object machines.

This is not to claim that the use of simple programming languages will solve the software problems. If that were true, machine languages would be ideal. Rather, the claim is that the problems cannot be solved with complex languages and that many of the current problems have been aggravated by the use of unnecessarily large and complex programming languages.

2. Readable/Understandable Programs

In the development of large software systems which must be integrated from many separately developed parts or software subsystems, have long lifetimes, and must go through many modifications to their functional requirements, it is essential that the programs be readable and understandable by their authors and maintainers. Only when the programmer thoroughly understands his own programs can he convince himself or anyone else of their correctness. We cannot accurately predict the effect of a program if we cannot understand it and we cannot modify, repair, or extend a program if we cannot predict the impact of changes.

3. Correct Translator

The programmer must have confidence in the compiler. The implementation must be consistent with the language semantics, it must report errors rather than compile a garbage object code, it must produce the object code a good programmer would expect, and it should not change the meaning of programs from time to time. More simply, it should be correct, consistent, and predictable. The language features it must implement, their form in the source language, and the quality of the source language definition affects the ability of the translator to meet these goals.

4. Error-Intolerant Translator

The issue here is, when are programming errors to be detected: during the design, during program development, during system validation and test, or while the program is in use? In

many DoD applications, errors discovered in operational use can have catastrophic consequences. System test and validation is an ideal time to build confidence in a system and to test the most common cases. It is, however, impossible to test every case, and there must be confidence that the limited tests employed are indicative of the total program reliability. Errors should be detected during the design and development phases. The translator can help by reporting all errors which it can detect. The number and importance of these will be small (*i.e.*, syntax only) if the source language is only a coding language. Reducing the syntactic choices of the user by restricting the set of acceptable program strings can increase the distance between correct programs and increase the probability that syntax errors will result in syntactically incorrect programs, but this is of very limited help. The important errors are semantic and can be detected by the compiler only if the programming language is a design and documentation language as well as a coding language. That is, if it allows specification of the programmers intent as well as his actions (*e.g.*, range and types of variables), it allows redundant specifications (*e.g.*, types of formal and actual parameters), it does not violate his intentions (*e.g.*, no implicit type conversions), permits him to identify the parts of the program in which a program component will be used (*e.g.*, scope of access specification), and allows him to deny access to nonessential properties of his data and programs. Each of these provides information which allows the translator to check the program design for semantic consistency and to verify that the programmer has, in fact, conformed to his own conventions and stated intent. These same specifications will also contribute to the readability and maintainability of the program.

The goal, of course, should not be to maximize the number of detectable errors, but rather to minimize the number of nondetectable errors, the difference being that the language should

be first concerned with the prevention of errors and secondly, with the detection of errors which cannot be prevented by the language. Many errors are prevented, for example, when the HOL does not permit run time modifications to executable code or does not permit Boolean operations on floating point values. In any case, the language design should attempt to minimize the kinds of errors which can occur and should attempt to maximize the number of those which are detectable by a translator. Finally, any translator for the HOL should report all errors which it can detect.

5. Efficient Object Code

Software should strive to make optimal use of all the resources associated with the design, development, use, and maintenance of the software. In some DoD software systems, the major costs are in hardware because of multiple deployment (*e.g.*, fire control) or are subject to computer hardware constraints because of the environment (*e.g.*, avionics). In some control systems, there are critical time constraints which are difficult for even machine language programs to meet; in some simulation problems, the full job is still beyond the capabilities of even the largest computers, and in some data processing applications, limited memory resources require shuffling of large quantities of data between main and peripheral memories and create a bottleneck at the I/O interface. In all these applications, the efficiency of program and/or data object representations can be very important. Optimal program design, of course, must be relative to some design criteria which are measured in terms of some resource, such as time, space, manpower, or dollars. Optimal program design does not imply, for example, that compile time resources should be wasted in squeezing out unneeded object efficiency.

C. CRITERIA TO ASSURE A COMMON PROGRAMMING LANGUAGE PRODUCT

1. Complete Source Language

Every user level aspect of the language should be specified in its defining documentation. None should be left to be made arbitrarily and uniquely by each translator, operating system, and object machine. The language proliferation problem stems primarily from development of evermore new incompatible versions of existing languages. In many cases, new languages are developed for sound reasons, but the effect is the same. In some cases, the new language is given a new name, in others, it retains the old name and becomes incompatible dialect. In many instances, it is not so much that the new version violates previous standards, but that the standards are so incomplete and ambiguous that commonality is impossible. Even worse, many programming language definitions and standards intentionally leave portions of the semantics unspecified with the intent that they will be provided by the translator. This may be necessary for the appearance of commonality when incompatible compilers for a language already exist, but certainly not for a new language. Commonality, in more than name, requires that the language specification be complete. Every decision made in the programming process should be made irrevocably in the language design or the choice should be given explicitly to the programmer.

This does not mean that a program must be implemented in the same way on all object machines, only that the resulting semantics be the same in all ways important to the program logic. The user should not have to overspecify his programs; he should be able to leave *don't care* and *don't-care-within-limits* conditions to the translator. For example, he might be able to specify the minimal numeric precision required by his program with the exact implementation determined by the translator and object machine. The order of evaluation of terms in

an expression or of the operators in a sequence of associative operators should be left to the translator when it does not affect the computation.

2. Wide Applicability

The wide use of a very small number of programming languages is desirable for many reasons. Training costs are reduced and personnel become more versatile. Project costs should be less, because existing software can be reused. Programming costs should be lower because funds can be expended on improving existing software tools and building more powerful tools. Increasingly, applications are not pure; they may be primarily numerical computation, report generation, sensor processing, process control, file searching, etc., but each has ingredients of several other applications. Special-purpose, problem-oriented languages lack the generality and adaptability to grow with the applications. Confidence that the next project or assignment will use the same language creates incentives at both the management and programmer level to develop flexible and reusable software.

3. Implementable

A programming language will be widely used only if it is capable of inexpensive translation into object computer programs. If the language is simple and easy to implement, the cost of implementation will be lower and translators will be more widely available. Potential users will like it and want to use it only if the cost in machine resources and elapsed time for translation is reasonable. The smaller the translator and the smaller the machines which can host the translator, the larger the number of users.

4. Static Design

There can be no commonality if the programming languages, are constantly changing. Projects often develop their own

compilers. These compilers do not implement exactly some existing source language, but are extended subsets which attempt to incorporate the latest software technology and special features useful to their project while omitting seldom-used features. This approach, while providing specialized tools sometimes well-suited to the task at hand, increases the research content, risk, and cost of the project. The alternative is to draw a distinct line between research in programming languages and engineering development of a language. A language can be built as an engineering development, incorporating the current state of the art but not going beyond it; its design can be frozen and the language used in that form for an extended period. A willingness to freeze languages and to accept the best technology of some past moment is essential to obtain the benefits of commonality. Research on software technology, management, language features, and language design should continue in parallel with use of a common language. Growth and improvement in production programming languages should be limited to discrete, clearly defined points when there are major improvements to be incorporated rather than on a continuous basis.

A static design cannot be maintained without controls. Both implicit and explicit controls will probably be needed. Explicit controls might include language standards, configuration management of language implementations, and policy requiring use of the common language. Implicit controls are at least as important. They might include economic incentives, such as low cost access to existing support software, software development aids and application packages, lower-risk developments, and greater availability of trained programming personnel.

5. Reusability

A common language alone, even if it has easily accessible, compatible, and efficient implementations, is insufficient to encourage the development of flexible and reusable software. Reusability does not result merely from the use of a common language. A major problem in writing reusable software is that the generality required for reusability precludes it from being acceptably efficient in many applications. General-purpose routines will be widely used only if it is easy to tailor them to efficient, special-purpose variants. Most desirably, these specializations would be made automatically by the compiler when constant arguments are used, or semiautomatically, as when the programmer specifies that a call is to be compiled as an open, rather than closed, subroutine. Language features should be chosen to encourage the development and use of reusable software.

6. A Pedagogical Language

A good pedagogical programming language is one which is easy to learn and well suited to teaching programming methodology and techniques. In applications for which there is currently no common language, selection of a common easy-to-learn language will reduce the difficulty and cost of adopting a common language. A language well-suited for teaching and learning programming methodology and techniques is, of course, also well-suited for applying those methods and techniques.

Already, in the short time of this effort, unsolicited interest in using a common DoD language has been shown by universities. They not only need a modern pedagogical language, but also one which has many users outside the academic community. Few, if any, of the commercial and academic programming languages satisfy both requirements.

V. THE NEEDED CHARACTERISTICS

The set of characteristics prescribed below represents a synthesis of the requirements submitted by the Military Departments and is intended to be consistent with the language criteria of Section IV, self-consistent, and achievable with existing computer software and hardware technology. The needed characteristics are the requirements to be satisfied by an existing, modified, or new language selected as a common language. The characteristics prescribe capabilities and properties which a common DoD language should possess, but are not intended to impose any particular language features or mechanization of those capabilities.

The large number of characteristics reflects an attempt at thoroughness in dealing with the relevant issues. Similarly, the length of the discussion for many items reflects the need to resolve the ambiguities, examine the implications, and demonstrate the feasibility of the compendious statement introducing that characteristic. Because the characteristics address issues in the design, implementation, and use of the language and properties of the resulting product, there should be no correlation between the number of characteristics discussed here and the number of features in a language which satisfies these characteristics. Many of the characteristics will influence the choice of many features, and every feature will be influenced by many of the needed characteristics that good language design is a unification process. Any language that satisfies these characteristics must be smaller and simpler than the set of issues underlying its choice.

The header of each item gives a general description of the needed language characteristic, while the subsequent paragraph(s) of its body provide clarification, discuss some of the implications and problems, provide the rationale behind its inclusion, and further detail the requirement. The entire text, not just the headers, constitutes the requirements.

A. DATA AND TYPES

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

By the type of a data object is meant the set of objects themselves, the essential properties of those objects, and the set of operations which give access to and take advantage of those properties. The author of any correct program in any programming language must, of course, know the types of all data and variables used in his programs. If the program is to be maintainable, modifiable, and comprehensible by someone other than its author, then the types of variables, operations, and expressions should be easily determined from the source program. Type specifications in programs provide the redundancy necessary to verify automatically that the programmer has adhered to his own type conventions. Static-type definitions also provide information at compile time necessary for production of efficient object code. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record

structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean, and character, and as type generators, will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type).

These are the common data types and type generators of most programming languages and object machines. They are sufficient, when used with a data definition facility (see D6, E6, and J1), to mechanize other desired types (*e.g.*, complex or vector) efficiently.

A3. The source language will require global (to a scope) specification of the precision for floating-point arithmetic and will permit the global precision to be overridden by precision specification for individual variables. These specifications will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

This is a specification of what the program needs, not what the hardware provides. Machine independence, in the use of approximate value numbers (usually with floating-point representation), can be achieved only if the user can place constraints on the translator and object machine without forcing a specific mechanization of the arithmetic. Precision specifications, as the maximum required by the object code, provide all the power and guarantees needed by the programmer, without unnecessarily constraining the object machine realization.

Precision specifications will not change the type of reals or the set of applicable operations. Precision specifications apply to arithmetic operations as well as to the data, and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program. Specification of the precision will also contribute to the legibility and implementability of programs.

A4. Fixed-point numbers will be treated as exact quantities which have a range and a fractional step size determined by the user at compile time. Scale-factor management will be done by the compiler.

Scaled integers are useful approximations to real numbers when dealing with exact quantity fractional values, when the object machine does not have floating-point hardware, and when greater precision is required than is available with the floating-point hardware. Integers will also be treated as exact quantities, with a step size equal to one.

A5. Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the order of characters and their literal form to be used in programs. These properties of a character set would be unalterable at run time. The definition of a character set should reflect on the manner it is used within a program and not necessarily on the print representation a particular physical device associates with a bit pattern at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as USASCII and EBCDIC will be available in a standard library. Note that access to a linear array filled with the characters of an alphabet, A, and indexed by an alphabet, B, will convert strings of characters from B to A.

A6. The language will require user specification of the number of array dimensions, the range of subscript values for each array dimension, and the type of each array component. The number of dimensions, the type, and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

This is general enough to permit both arrays which can be allocated at compile or load time and arrays which can be allocated at scope entry, but does not permit dynamic change to the size of constructed arrays. It is sufficient to permit allocation of space pools which the user can manage for allocation of more complex data structures, including dynamic arrays. The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This provides all that is safe to use in CMS-2 and JOVIAL OVERLAY and in FORTRAN EQUIVALENCE. It permits hierarchically structured data of heterogeneous type, permits records to have alternative structures, as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory or the

dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record, but should be any Boolean expression.

B. OPERATIONS

B1. Assignment and reference operations will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of encapsulated type definitions (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion.)

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

Equivalence is an essential universal operation which should not be subject to restriction on its use. There are many useful equivalence operations for some types, and a language definition cannot foresee all these for user-defined types. Equivalence, meaning logical identity, and not bit-by-bit comparison on the internal data representation, however, is required for all data types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (*i.e.*, numbers, characters, Boolean values, and types defined by enumeration) and that elements of disjoint

types never be identical. Consequently, its usefulness at run time is restricted to data of the same type or to types with nonempty intersections. For floating-point numbers, identity will be defined as the same within the specified (minimum) precision.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

Numbers and types defined by enumeration have an obvious ordering which should be available through relational operations. All six relational operations will be included. It will be possible to inhibit ordering definitions when unordered sets are intended.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed-point arguments and remainder), and negation.

These are the most widely used numeric operations and are available as hardware operations in most machines. Floating-point operations will be precise to at least the specified precision.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least-significant digits and will never be implicit for integers and fixed-point numbers. Implicit rounding beyond the specified precision will be allowed for floating-point numbers.

These requirements seem obvious, particularly for floating-point numbers, and yet many of our existing languages truncate the most significant mantissa digits in some mixed and floating-point operations.

B6. The built-in Boolean operations will include and, or, not, and xor. Operations such as and and or on scalars will be evaluated in short-circuit mode.

Short-circuit mode as used here is a semantic rather than an implementation distinction and means that *and* and *or* are, in fact, control operations which do not evaluate side effects of their second argument if the value of the first argument is *false* or *true*, respectively. Short-circuit evaluation has no disadvantages over the corresponding computational operations, sometimes produces faster executing code in languages where the user can rely on the short-circuit execution, and improves the clarity and maintainability of programs by permitting expressions such as, $i \leq ?\ \&\ A[i] > x$, which could be erroneous were short-circuit execution not intended. Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or, respectively.

F7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one-for-one compatibility in type. Correspondence will be by position in similarly shaped arrays. In many situations, component-by-component operations are done on array elements. In fact, a primary reason for having arrays is to permit large numbers of similarly treated objects to

have a uniform notation. Operations on large data aggregates available directly in the source language hide the details of the sequencing and, thereby, simplify the program and make more optimizations available. In addition, they permit simultaneous execution on machines with parallel processing hardware. Although component-by-component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. A matrix might be defined using an array, but it will not inherit the array operations automatically. Multiplication for matrices would, for example, be unnatural, confusing, and inconvenient if the product operator for matrices were interpreted as a component-by-component operation instead of cross product of corresponding row and column vectors. Component-by-component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

Transfers between arrays or records of identical logical structure are necessary to permit efficient run time conversion from one object representation to another, as might be done when data is packed to reduce peripheral storage requirements and I/O transfer times, but need to be unpacked locally to minimize processing costs.

B8. There will be no implicit type conversions, but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed-point, and floating-point data, between the object representation of numbers and their representations as characters, and between fixed-point scale factors.

Implicit-type conversions, which represent changes in the value of data items without an explicit indicator in the program, are not only error prone but can lead to unexpected run time overhead.

B9. Explicit conversion operations will not be required between numeric ranges. There will be a run time exception condition when any integer or fixed-point value is truncated.

Because ranges do not form closed systems, range validation is not possible at compile time (*e.g.*, I:=I+1 may be a range error). At best, the compiler might point out likely range errors. (This requirement is optional for hardware installations which do not have overflow detection.)

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to assign and reassign I/O devices dynamically, will provide user control for exception conditions, and will not be installation-dependent.

Whether the referenced "files" are real or virtual and whether they are hardware devices, I/O channels, or logical files, depends on the object machine configuration and on the details of its operating system, if present. In any programming system, I/O operations ultimately reduce to sending or receiving data or control information to a file or to a device controller. These can be made accessible in an HOL in an abstract form through a small set of generic I/O operations (like *read* and *write*, with appropriate device and exception parameters). Note that devices and files are similar in many

respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

As with any data type, power sets will be useful only if there are operations which can create, select, and interrogate them. Note that this provides only a very special class of sets, but one which is very useful for computations on sets of indicators, flags, and similar devices in monitoring and control applications. More general sets, if desired, must be defined, using the type definition facilities.

C. EXPRESSIONS AND PARAMETERS

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This is a semantic restriction on the evaluation order of arguments to expressions. It provides an explicit rule (i.e., left-to-right) for order of argument evaluation, but allows the implementations to alter the actual order in any way which does not change the effect. This provides the user with a simple rule for determining the effects of interactions among argument evaluations without imposing a strict rule on compilers which are sophisticated enough to detect potential side-effects and optimize through reordering of arguments when the evaluation

order does not affect the result. Control operations (*e.g.*, conditional and iterative control structures), of course, must be exceptions to this general rule, since control operations are, in fact, those operations which specify the sequencing and evaluation rules for their arguments.

> *C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.*

The operator/operand structure of expressions must not be psychologically ambiguous (*i.e.*, to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules, by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (*e.g.*, $X+Y+Z$ *and* $X+Y\times Z$). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

> *C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.*

This is an example of not imposing arbitrary restrictions and special case rules on the user of the source language. Special mention is made here only because so many languages do restrict the form of expressions. FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C4. Constant expressions will be allowed in programs wherever constants are allowed, and constant expressions will be evaluated before run time.

The ability to write constant expressions in programs has proven valuable in languages with this capability, particularly with regard to program readability and in avoiding programmer error in externally evaluating and transcribing constant expressions. They are most often used in declarations. There is no need, however, for constant expressions to impose run time costs for their evaluation. They can be evaluated once at compile time, or if this is inconvenient because of incompatibilities between the host and object machines, the compiler can generate a code for their evaluation at load time. In any case, the resulting value should be the same (at least within the stated precision), regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness, and maintainability of programs, and does not impose any run-time costs.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, types, exception handling, parallel processes, declarations, or built-in operations. There will be no special operations (e.g., array substructuring) applicable only to parameters.

Uniformity and consistency contribute to ease of learning, implementing, and using a language; allow the user to concentrate on the programming task instead of the language; and lead to more readable, understandable, and predictable programs.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Type transfers hidden in procedure calls with incompatible formal and actual parameters, whether intentional or accidental, have long been a source of program errors and of programs which are difficult to maintain. On the other hand, there is no reason why the subscript ranges for arrays cannot be passed as part of the arguments. Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

C7. There will be only four classes of formal parameters. For data, there will be those which act as constants, representing the actual parameter value at the time of call, and those which rename the actual parameter, which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and there will be a class for procedure parameters.

The first class of data parameter acts as a constant within the procedure body. Assignments cannot be made to these parameters and they cannot be changed during execution of the procedure. Their corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames

the actual parameter which must be a variable. The address of the actual parameter variable will be determined by (or at) the time of call and will be unalterable during execution of the procedure. Assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. These are the only two widely used parameter-passing mechanisms for data. The many alternatives (at least 10 have been suggested) add complexity and cost to a language without sufficiently increasing its clarity or power. A language with exception-handling capability must have a way to pass control and related data through procedure-call interfaces. Exception-handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional on the formal side (i.e., in the procedure declaration). None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time *type* parameters. This generic procedure capability, for example, allows the definition of stacks and queues and their associated operations on data of any given type, without knowing the data type when the operations are defined. This does not conflict with the requirement for compile-time-determinable type determination (A1), because the language permits union types (see E6) and compile time evaluation of constant expressions (see C4), including type testing expressions.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description, and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as *print*, generalizations of operations which are both commutative and associative, such as *max and min*, and repetitive application of the same binary operation such as the Lisp *list* operation. The use of operations with variable numbers of arguments need not and will not cause relaxation of any compile-time checks, require use of multiple-entry procedures, allow the number of actual parameters to vary at run time, or require special calling mechanisms. If the parameters which can vary are limited to a program-specified type treated as any other argument on the call side and as elements of an array within the procedure definition, full type checking can be done at compile time. There will be no prohibition on writing a special case of a procedure for a particular number of arguments.

D. VARIABLES, LITERALS, AND CONSTANTS

D1. The user will have the ability to associate constant values of any type with identifiers.

The use of identifiers to represent constant values has often made programs more readable, more easily modifiable, and less prone to error when the value of a constant is changed.

Associating constant values with an identifier is preferable to assigning the value to a variable, because it is then clearly marked in the program as a constant, can be automatically checked for unintentional changes, and often can have a more efficient object representation.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Constants are needed for all atomic data types and should be provided as part of the language definition for built-in types. Regardless of the source of the data and of the object machine, the value of constants should be the same. For integers, it should be exact, and for reals it should be the same, within the specified precision. Compiler writers, however, would disagree. They object to this requirement on two grounds: that it is too costly if the host and object machines are different, and that it is unnecessary if they are the same. In fact, all costs are at compile time and must be insignificant compared to the life-time costs resulting from object codes containing the wrong constant values. As for being unnecessary, there have been all too many cases of different values from program and data literals on the same machine because the compile time and run time conversion packages were different and imprecise.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

The ability to initialize variables at the time of their allocation will contribute to program clarity, but a requirement to do so would be an arbitrary and sometimes costly decision. Default initial values, on the other hand, contribute to neither program clarity nor correctness and can be even more costly at run time. It is usually a programming error if a variable is accessed before it is initialized. It is desirable that the translator give a warning when a path between the declaration and use of a variable omits initialization. Whether a variable will be assigned is, in general, an unsolvable problem, but it is sometimes determinable whether assignments occur on potential paths. In the case of arrays, it is possible at compile time only to determine that some components (but not necessarily which) have been initialized. There will be provision (at user option) for run time testing for initialization.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed-point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step-size specifications will not be interpreted as defining new types.

Range specifications are a special form of assertion. They aid in understanding and determining the correctness of programs. They can also be used as additional information by the compiler in deciding what storage and allocation to use (*e.g.*, half words might be more efficient for integers in the range 0 to 1000). Range specifications also offer the opportunity for the translator to insert range tests automatically

for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds checks, however, can be only as valid as the range specifications, which cannot, in general, be validated at compile time. Range specifications on approximate valued variables (usually with floating-point implementation) also offer the possibility of their implementation using fixed-point hardware.

D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

There should not be any arbitrary restrictions on the structure of data. This permits arrays to be components of records or arrays and permits records to be components of arrays.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Depending on the data type, variables of that type will hold values which either can be shared or must be unique to that variable. Assignment to a variable of a shared value type will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a variable of a unique value type will mean that the variable's name is to label a copy of the object being assigned.

For data without alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, whether values are shared or copies is meaningful only for composite types and for arrays and records with composite components. Whether a composite type has shared or copied values will be specified as part of the type definition. The use of pointers (*i.e.*, shared values) will be kept safe by prohibiting variables from holding values whose allocation scopes are narrower than that of the variable. Such a restriction is easily enforced at compile time using hierarchical scope rules, providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety, using a (user or library-defined) space pool which is either local (*i.e.*, own) or global to the type definition. If variables of a type are not shared, dynamic storage allocation will be required for assignment unless their size is constant and known at the time of variable allocation. Thus, copied variables will be permitted only for types (a) whose data have a structure and size which is constant in the type definition, or (b) which manage the storage for their data as part of the type definition. Because shared values are often less expensive at run time than copied values and are subject to fewer restrictions, the specification of copied values will be explicit in programs (this is similar to the ALGOL-60 issue concerning the explicit specification of *value* (*i.e.*, copied) and *name* (*i.e.*, shared). The need for pointers is obvious in building data structures with shared or recursive substructures, such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type- and access-restricted pointers will provide the power of general pointers, without their undesirable characteristics.

E. DEFINITION FACILITIES

E1. The user of the language will be able to define new data types and operations within programs.

The number of specialized capabilities needed for a common language is large and diverse. In many cases, there is no consensus as to the form these capabilities should take in a programming language. The operational requirements dictating specific specialized language capabilities are volatile, and future needs cannot always be foreseen. No language can make available all the features useful to the broad spectrum of military applications, anticipate future applications and requirements, or even provide a universally "best" capability in support of a single application area. A common language needs capability for growth. It should contain all the power necessary to satisfy all the applications and the ability to specialize that power to the particular application task. A language with defining facilities for data and operations often makes it possible to add new application-oriented structures and to use new programming techniques and mechanisms through descriptions written entirely within the language. Definitions will have the appearance and costs of features built into the language while they are actually catalogued as application packages. The operation definition facility will include the ability to define new infix and prefix operators (but see H2 for restrictions). No programming language can be all things to all people, but a language with data and operation definition facilities can be adapted to meet changing requirements in a variety of areas.

The ability to define data and operations is well within the state of the art. Operation definition facilities in the form of subroutines have been available in all general-purpose programming languages since at least the time of early FORTRANs.

Data definition facilities have been available in a variety of programming languages for almost 10 years and reached their peak with more than 30 extensible languages in 1968 and shortly thereafter (Ref. 4). A trend toward more abstract and less machine-oriented data specification mechanisms has appeared more recently in PASCAL (Ref. 5). Data type definitions, with operations and data defined together, are used in several languages, including SIMULA-67 (Ref. 6). On the other hand, there is currently much ferment as to what is the proper function and form of data type definitions.

E2. The use of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no *ad hoc* special cases or inconsistent rules to interfere with and complicate learning, using, and implementing the language. If built-in features and user-defined data structures and operations are treated in the same way throughout the language, so that the base language, standard application libraries, and application programs are treated in a uniform manner by the user and by the translator, then these distinctions will grow dim, to everyone's advantage. To achieve these goals, full encapsulation capabilities are needed, as well as ways to specify special selection, printing, and storage management policies for underlying representations. When the language contains all the essential power, when few can tell the difference between the base language and library definitions, and when the introduction of new data types and routines does not have an impact on the complier and the language standards, then there is little incentive to proliferate languages. Similarly, if type definitions are processed entirely at compile time and the language allows full program specification of the internal representation, there need be no penalty to run time efficiency for using defined types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

As programmers, we should not expect the translator to write our programs for us (at least in the immediate future). If we somehow know that the translator's default convertion is compatible with our needs for the case at hand, we should still document the choice so others can understand and maintain our programs. Neither should we be able to delay definitions (possibly forget them) until they cause trouble in the operational system. This is a special case of requirement I1.

E4. The user will be able, within the source language, to extend existing operators to new data types.

When an operation is an abstraction of an existing operation for a new type or is a generalization of an existing operation, it is inconvenient, confusing, and misleading to use any but the existing operator symbol or function name. The translator will not assume that commutativity of built-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The data objects are given a representation in terms of existing data structures, but they are of little value until operations are available to take advantage of their special properties. When one obtains access to a type, he needs its operations as well as its data. Numeric data is needed in many applications, but is of little value without arithmetic operations. The definable operations will include constructors, selectors, predicates, and type conversions.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

The above list comprises a currently known set of useful definitional mechanisms for data types which do not require run time support, as do garbage collection and dynamic storage allocation. In conjunction with pointers (see D6), they provide many of the mechanisms necessary to define recursive data structures, and efficient sparse data structures.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Free union adds no new power not provided by discriminated union, but does require giving up the security of types in return for programmer freedom. Range and subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and

often do not form a closed system under the superset operations. Unlike types, membership in subsets can be determined only at run time.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (*i.e.*, in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

F. SCOPES AND LIBRARIES

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases, the user may desire that each use of a defined program structure be independent (*i.e.*, the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F2. The ability to limit access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access specified in a type definition is necessary to guarantee that changes to data representations and to management routines which purportedly do not affect the calling programs, are, in fact, safe. By rigorously controlling the set of operations applicable to a defined type, the type definition guarantees that no external use of the type can accidentally or intentionally use hidden nonessential properties of the type. Renaming separately defined programming components is necessary to avoid naming conflicts when they are used.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and costly in creating large systems which are composed of many subsystems, because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F3. The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope, and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded, with the most local definition applying when the same identifer appears

at several levels. The language will use the above lexically embedded scope rules for declarations and other definitions of identifiers to make them easy to recognize and to avoid errors and ambiguities from multiple use of identifiers in a single scope.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

A simple base alone is not sufficient for a common language. Even though, in theory, such a language provides the necessary power and the capability for specialization to particular applications, the users of the language cannot be expected to develop and support common libraries under individual projects. There will be broad support for libraries common to users of well-recognized application areas. Application libraries will be developed as early as possible.

F5. Program components not defined within the current program and not in the base language will be maintained in libraries accessible at compile time. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

The usefulness of a language derives primarily from the existence and accessibility of specialized application-oriented data and operations. Whether a library should contain source or object codes is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. It should be remembered, however, that interfaces cannot be validated at program assembly time without some equivalent of their source language interface specifications, that

object modules are machine-dependent and, therefore, not portable, that source code is often more compact than object code, and that compilers for simple languages can sometimes compile faster than a loader can load from relocatable object programs. Library routines written in other languages will not be prohibited, provided the foreign routine has object codes compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (*e.g.*, parameter types, form, and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

> *F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems, through projects, to individual programs. There will be many specialized compools or libraries, any user-specified subset of which is immediately accessible from a given program.*

Compools have proven very useful in organizing and controlling shared data structures and shared routines. A similar mechanism will be available to manage and control access to related library definitions.

> *F7. The source language will contain standard machine-independent interfaces to machine-dependent capabilities, including peripheral equipment and special hardware.*

The convenience, ease of use, and savings in production and maintenance costs resulting from using high-order languages come from being able to use specialized capabilities without building them from scratch. Thus, it is essential that high-level capabilities be supplied with the language. The idea is

not to provide all the many special cases in the language, but to provide a few general cases which will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present, the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

G. CONTROL STRUCTURES

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives, each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. The object machine determines whether parallel processing is real (*i.e.*, by multiprocessing) or is synthesized on a single sequential processor, but if programs are written as if there is true parallel processing (and no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control

mechanisms within programs. This means that the capabilities of control mechanisms must be separable, so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the ALGOL-60 *for* would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition, and that the condition be tested before each iteration. Consequently, *for* cannot be composed to build other useful iterative control structures (*e.g.*, FORTRAN *do*). The ability to compose control structures does not imply an ability to define new control operations, and such an ability is in conflict with the limited parameter-passing mechanisms of C7.

G2. *The source language will provide a go-to operation applicable to program labels within its most local scope of definition.*

The *go to* is a machine-level capability which is still needed to fill in any gaps that might remain in the choice of structured control primitives, to provide compatibility for transliterating programs written in older languages, and because of the wide familiarity of current practitioners with its use. The language should not, however, impose unnecessary costs for its presence. The *go to* will be limited to explicitly specified program labels at the same scope level. Neither should the language provide specialized facilities which encourage its use in dangerous and confusing ways. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the *go to* and do not refer to *case* statements which are a general form for conditionals (see G3). This requirement should not be interpreted to conflict with the specialized form of control transfer provided by the exception-handling control structure of G7.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (*e.g.*, an *else* clause must follow each *if then*) so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation [*e.g.*, Zahn's device (Ref. 7) provides a good solution to the general problem]. Special mechanisms are also needed for the more common cases of the Boolean expression (*e.g., if then else*) and for value or type discrimination (*e.g., case* on one of a set of values or subtype of a union).

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run time execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. Specialized control structures (*e.g., While do*) have been used for the common situation in which the termination condition precedes each iteration. The most common

case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (*e.g.*, FORTRAN *do* or ALGOL-60 *for*). A problem which arises in many programming languages is that loop control variables are global to the iterative control, and, thus, will have a value after loop termination, but that value is usually an accident of the implementation. Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity, but must be an arbitrary decision which will not aid program clarity or correctness, and may interfere with the generation of efficient object codes. Loop control variables are, by definition, variables used to control the repetitive execution of a programmed loop, and, as such, will be local to the loop body. At loop termination, it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G5. *Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.*

Recursion is desirable in many applications because it contributes directly to their elegance and clarity and simplifies proof procedures. Indirectly, it contributes to the reliability and maintainability of some programs. Recursion is required to avoid unnecessarily opaque, complex, and confusing programs when programs operate on recursive data structures. Recursion has not been widely used in DoD software because many programming languages do not provide recursion, practitioners are not familiar with its use, and users fear that its run time costs are too high. Of these, only the run time cost would justify its exclusion from the language.

A major run time cost often attributed to recursion is the need for the presence of a set of "display" registers which are used to keep track of the addresses of the various levels of lexically-embedded environments and which must be managed and updated at run time. The display, however, is necessary only in programs in which routines access variables which are global to their own definition, but local to a more global recursive procedure. This possibility can easily be removed by prohibiting the definition of procedures within the body of a recursive procedure. The utility of such a combination of capabilities is very questionable, and this single restriction will eliminate all added execution costs for nonrecursive procedures in programs which contain recursive procedures.

As with any other facility of the language, routines should be implemented in the most efficient manner consistent with their use and the language should be designed so that efficient implementations are possible. In particular, the most efficient implementation for nonrecursive routines should be possible, regardless of whether the language or even the program contains recursive procedures. When any routine makes a procedure call as its last operation before exit (and this is quite common for recursive routines) the implementation might use the same data area for both routines and do a jump to the head of the called procedure, thereby saving much of the overhead of a procedure call and eliminating a return. The choice between recursive and nonrecursive routines involves trade-offs. Recursive routines can aid program clarity when operating on recursive data, but can detract from clarity when operating on iterative data. They can increase execution time when procedure call overhead is greater than loop overhead and can decrease execution times when loop overhead is the more expensive. Finally, program storage for recursive routines is often only a small fraction of that for a corresponding iterative procedure,

but the data storage requirements are often much greater because of the simultaneous presence of several activations of the same procedure.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

A parallel processing capability is essential in embedded computer applications. Programs must send data to, recieve data from, and control many devices which are operating in parallel. Multiprogramming (a form of pseudo-parallel processing) is necessary so that many programs within a system can meet their differing real time constraints. The parallel processing capability will minimally provide the ability to define and call parallel processes and the ability to gain exclusive use of system resources in the form of data structures, devices, and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. As with recursive routines,

most accessing and storage-management problems can be eliminated by prohibiting complex interactions with other language facilities where the combination has little, if any, utility. In particular, it will not be possible to define a parallel routine within the body of a recursive routine and it will not be possible to define any routine, including parallel routines, within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process, then it might require the user to give an upper bound on the number which can exist simultaneously. The latter requirement is reasonable for parallel processes because it is information known by the programmer and necessary to the maintainer, because parallel processes cannot normally be stacked, and because it is necessary for the compilation of efficient programs.

G7. The exception-handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

It is essential in many applications that there be no program halts beyond the user's control. The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception-handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user-defined exceptions, and any run-time-detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception-handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope

of the program. The transfers of control will, at the user's option, be either forward in the program (but never to a narrower scope of access or out of a procedure through its lexical structure) or out of the current procedure through its dynamic (*i.e.*, calling) structure. The latter form requires an exception-handling formal parameter class (see C7).

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, and which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo-parallel paths appear in a program, it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program. These situations should include the elapse of program-specified time intervals, occurrence of hardware interrupts, and those designated in the program. There will be no implicit evaluation of program-determined situations. Time delays will be program-specifiable for both real and simulated times.

H. SYNTAX AND COMMENT CONVENTIONS

H1. The source language will be free format with an explicit statement delimiter, will allow the use of memonically significant identifiers, will be based on conventional forms, will have a simple, uniform, and easily parsed grammar, will

not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

Clarity and readability of programs will be the primary criteria for selecting a syntax. Each of the above points can contribute to program clarity. The use of free format, mnemonic identifiers, and conventional forms allows the programmer to use notations which have their familiar meanings, to put down his ideas and intentions in the order and form that humans think about them, and to transfer skills he already has to the solution of the problem at hand. A simple, uniform language reduces the number of cases which must be dealt with by anyone using the language. If programs are difficult for the translator to parse, they will be difficult for people. Similar things should use the same notations with the special-case processing reserved for the translator and object machine. The purpose of mnemonic identifiers and key words is to be informative and increase the distance between lexical units of programs. This does not prevent the use of short identifiers and short key words.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms, or define new operator precedences.

If the user can change the syntax of the language, he can change the basic character and understanding of the language. The distinction between semantic extensions and syntactic extensions is similar to that between being able to coin new words in English or being able to move to another natural

language. Coining words requires learning those new meanings before they can be used, but at the same time increases the power of the language for some applications. Changing the grammer, (*e.g.*, Franglis, the use of French grammar with interspersed English words) however, undermines the basic understanding of the language itself, changes the mode of expression, and removes the commonalities which obtain between various specializations of the language. Growth of a language through definition of new data and operations and the introduction of new words and symbols to identify them is desirable, but there should be no provision for changing the grammatical rules of the language. This requirement does not conflict with E4 and does not preclude associating new meaning with existing operators.

H3. The syntax of source-language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64-character ASCII subset.

A common language should use notations and a character set convenient for communicating algorithms, programs, and programming techniques among its users. On the other hand, the language should not require special equipment (*e.g.*, card readers and printers) for its use. The use of the 64-character ASCII subset will make the language compatible with the federal information processing standard 64-character set, FIPS-1, which has been adopted by the U.S.A. Standard Code for Information Interchange (USASCII). The language definition will specify the translation from the publication language into the restricted character set.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Lexical units of the language should be defined in a simple, uniform, and easily understood manner. Some possible break characters are the space (*i.e.*, any number of spaces or end-of-line), the underline, and the tilde (Refs. 8 and 9). The space cannot be used if identifiers and user-defined infix operators are lexically indistinguishable, but in such a case, the formal grammar for the language would be ambiguous (see H1). A literal break character contributes to the readability of programs and makes the entry of long literals less error-prone. With a space as a break character, one can enter multipart (*i.e.*, more than one lexical unit) identifiers such as *REAL TIME CLOCK* or long literals, such as, *3.14259 26535 89793*. Use of a break can also be used to guarantee that missing quote brackets on character literals do not cause errors which propagate beyond the next end-of-line. The language should require separate quoting of each line of a long literal:

"This is a long"
"literal string".

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

Many elementary input errors arise at the end of lines. Programs are input on line-oriented media, but the concept of end-of-line is foreign to free-format text. Most of the error-prone aspects of end-of-line can be eliminated by not allowing lexical units to continue over lines. The sometimes undesirable

effects of this restriction can be avoided by permitting identifiers and literals to be composed from more than one lexical unit (see H4) and by evaluating constant expressions at compile time (see C4).

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms, such as are used for control structures and declarations, or they are used as infix operators, or as some form of parenthesis. To avoid confusion and ambiguity, key words will be reserved, that is, not used as identifiers. Key words will be few, because each new key word introduces another case in the parsing rules, adding to the complexity of the language, also, too many key words inconvenience and complicate the programmer's task of choosing informative identifiers. Key words should be concise, but it is more important that they be informative than short. A major exception is the key word introducing a comment; in this case, the comment, not its key word, should do the informing. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H7. The source language will have a single, uniform comment convention. Comments will be easily distinguishable from code, will be introduced by one or possibly two language-defined characters, will permit any combination of characters to appear, will be able to appear at any reasonable point in a program, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

These are all obvious points that will encourage the use of comments in programs and avoid their error-prone features in some existing languages. Comments at any reasonable point in a program will not be taken to mean that they can appear internally in a lexical unit, such as an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H8. The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more *BEGINs* than *ENDs*, the translator might insert enough *ENDs* at the end of the program to make up the difference. This makes programs easier to write because it sometimes saves the programmer writing several *ENDs* at the end of programs and because it eliminates all syntax errors for missing *ENDs*. Failure to require proper parentheses-matching makes it more difficult to write correct programs. Good programming practice requires that

matching parentheses be included in programs, whether or not they are required by the language. Unfortunately, if they are not required by the language, there can be no syntax check to discover where errors were made. The language will require full parentheses-matching. This does not preclude syntactic features such as *case x of* s_1, $s_2 \ldots s_n$ *end case* in which *end* is paired with a key word other than *begin*. Nor does it, alone, prohibit open forms such as *if-then-else-*.

H9. There will be a uniform referent notation.

The distinction between function calls and data reference is one of representation, not of use. Thus, there will be no language-imposed syntactic distinction between function calls and data selection. If, for example, a computed function is replaced by a lookup table, there should be no need to change the calling program. This does not preclude the inclusion of more than one referent notation.

H10. No language-defined symbols appearing in the same context will have essentially different meanings.

This contributes to the clarity and uniformity of programs, protects against psychological ambiguity, and avoids some error-prone features of extant languages. In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/1 subroutines), and would exclude the use of an assignment operator for other than assignment (*e.g.*, left-hand-side function calls). It would not, however, require different operator symbols for integer, real, or even matrix arithmetic, since these are, in fact, special cases of the same abstract operations, and would allow the use of generic functions applicable to several data types.

I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined, or explicitly in each program.

The only alternative is implementation-dependent defaults, with the translator determining the meaning of programs. What a program does should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit: in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

The language should provide a high degree of management control and visibility to programs and self-documenting programs, with the programmer required to make his decision explicit. On the other hand, the programmer should not be forced to overspecify his programs and thereby cloud their logic, unnecessarily eliminate opportunities for optimization, and misrepresent arbitrary choices as essential to the program logic.

Defaults will be allowed, in fact encouraged, in "don't care" situations. Such defaults will include data representations (see J4), open *vs.* closed subroutine calls (see J5), and reentrant *vs.* nonreentrant code generation.

I3. The user will be able to associate compile-time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines, and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object-machine configuration. The user should have control over the compile-time variables used in his program. Typically, they would be associated with the object computer model; memory size; special hardware options; operating system, if present; peripheral equipment; or other aspects of the object-machine configuration. Compile-time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile-time variables. In such cases, the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object-machine configuration or of other compile-time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general-purpose capability for conditional compilation.

I5. The source language will contain a simple, clearly identifiable base which houses all the power of the language. To the extent possible, the base will be minimal, with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

The capabilities available in any language can be partitioned into two groups, those definable within the base, and those providing an essential primitive capability of the language. The smaller and simpler the base, the easier the language will be to learn and use. A clearly delineated base, with features not in the base defined in terms of the base, will improve the ease and efficiency of learning, implementing, and maintaining the language. Only the base need be implemented to make the full source-language capability available.

Base features will provide relatively low-leveled general-purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating specialized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language, using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language, but not necessarily with the same object efficiency.

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. In each case, however, (a) some limit must be set, (b) whatever the limit, it will affect some program and therefore must be known by the users of the translator, (c) letting each translator set its own limits means that programs will not be portable, (d) setting the limits very high requires that the translator be hosted only on large machines, and (e) quite low limits do not impose significantly on either the power of the language or the readability of programs. Thus, the limits should be set as part of the language definition. They should be small enough that they do not dominate the compiler, and large enough that they do not interfere with the usefulness of the language. If they were set at, say, the 99-percent level, based on statistics from existing DoD computer programs, the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run-time storage, access to specialized peripheral equipment, use of special hardware capabilities, and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The base language and library definitions might contain features and capabilities not needed by everyone, or not by everyone all the time. The language should not force programs to require greater generality than they need. When a program does not use a feature or capability, it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as automatic and dynamic array allocation, process scheduling, file management, and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language. The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main storage and backing storage which is not under program control (unless the object machine has virtual memory with underlying management beyond the control of all its users). Language features will result in special efficient object code when their full generality is not used. A large number of special cases should compile efficiently. For example, a program performing numeric calculations on unsubscripted real variables should produce code no worse than FORTRAN. Parameter-passing for single-argument routines might be implemented much less expensively than multiple-argument routines.

One way to reduce costs for unneeded capabilities is to have a base language whose data structures and operations provide a single capability which is composable and has a straightforward implementation in the object code of conventional

architecture machines. If the base language components are easily composable, they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability, they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory, they will have near-minimum or at least low-cost implementation on many object machines.

J2. Any optimizations performed by the translator will not change the effect of the program.

More simply, the translator cannot give up program reliability and correctness, regardless of the excuse. Note that for most programming languages, there are few known safe optimizations and many unsafe ones. The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This allows optimization by code motion, providing that motion does not change the effect of the program.

J3. The source language will provide encapsulated access to machine-dependent hardware facilities, including machine language code insertions.

It is difficult to be enthusiastic about machine language insertions. They defeat the purpose of machine independence; constrain the implementation techniques; complicate the diagnostics; impair the safety of type checking; and detract from the reliability, readability, and modifiability of programs. The use of machine language insertions is particularly dangerous in multiprogramming applications, because they impair the ability to exclude, *a priori*, a large class of time-dependent

bugs. Rigid enforcement of scope rules by the compiler in real-time applications is a powerful tool to ensure that one sequential process will not interfere with others in an uncontrolled fashion. Similarly, when several independent programs are executed in an interleaved fashion, the correct execution of each may depend on the others not improperly using machine language insertions.

Unfortunately, machine language insertions are necessary for interfacing special-purpose devices, for accessing special-purpose hardware capabilities, and for certain code optimizations on time-critical paths. Here we have an example of Dijkstra's dilemma (see Chapter I, Section B), in which the mismatch between high-level language programming and the underlying hardware is unacceptable and there is no feasible way to reject the hardware. The only remaining alternative is to "continue bit pushing in the old way, with all the known ill effects". Those ill effects can, however, be constrained to the smallest possible perimeter, in practice, if not in theory. The ability to enter machine language should not be used as an excuse to exclude otherwise-needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone, and not dependent on the translator characteristics. Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine-language insertions will be permitted only within the body of compile time conditional statements (see I4), which depend on the object-machine configuration (see I3). They will not be allowed to be interspersed with executable statements of the source language.

J4. It will be possible within the source language to specify the object representation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space tradeoff to the translator. If not specified, the object representation will be optimal, as determined by the translator.

It is often necessary to give detailed specifications of the object data representations to obtain maximum density for large data files, to meet format requirements imposed by the hardware or peripheral equipment, to allow special optimizations on time-critical paths, or to ensure compatibility when transferring data between machines.

It will be possible to specify the order of fields, the width of fields, the presence of "don't care" fields, and the position of word boundaries. It will be possible to associate source-language identifiers (data or program) with special machine addresses. The use of machine-dependent characteristics of the object representation will be restricted, as with machine-dependent code (see J3). When multiple fields per word are specified, the compiler may have to generate some form of shift and mask operations for source-program references and assignments to those variables (*i.e.*, fields). As with machine-language insertions, object data specifications should be used sparingly and the language features for their use must be Spartan.

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator. The translator might, for example, attempt to minimize access time and/or

memory space in determining the object representation. It might, depending on the object-machine characteristics, assign variables and fields of records to full words, but assign array elements to the smallest of bits, bytes, half words, words, or exact-multiple words permitted by the logical description.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

The use of inline open procedures can reduce the run time execution costs significantly in some cases. There are the obvious advantages in eliminating the parameter passing, in avoiding the saving of return marks, and in not having to pass arguments to and from the routine. A less obvious, but often more important, advantage in saving run time costs is the ability to execute constant portions of routines at compile time and, thereby, eliminate time and space for those portions of the procedure body at run time. Open routine capability is especially important for machine-language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (*i.e.*, read only) parameter in the body have the same value. If a routine is not specified as either open or closed, the choice will be optimal (with respect to space or time) as determined by the translator.

VI. CHARACTERISTICS NEEDED FOR OTHER ASPECTS OF THE COMMON-LANGUAGE EFFORT

The material reported in this chapter was generated by the Services at the same time as the technical characteristics described in the preceding Chapter but is concerned with the translators, support software, documentation, training, standards, application libraries, management policy, and procurement practices for the common language and its use. These issues are important, while mistakes and oversights in the technical characteristics can guarantee failure of the common-language effort, success is not guaranteed, no matter how technically meritorious the resulting language. Success can only be guaranteed by close attention to a variety of nontechnical issues, including those considered below.

Several of these issues, including those of implementation, documentation, and support will either directly or indirectly affect the acceptability of candidate languages. As with the needed technical characteristics for the common language, the issues raised here are often not resolved at the most detailed level. Until more detailed characteristics of the language come into focus, there is no rationale with which to resolve all these issues in detail.

A. PROGRAM ENVIRONMENT

K1. The language will not require that the object machine have an operating system. When the object machine does have an operating system or executive program, the hardware/operating system combination will be interpreted as defining an abstract machine which acts as the object machine for the translator.

A language definition cannot dictate the architecture of existing object machines, whether defined entirely in hardware or in a hardware/software combination. It can provide a source-language representation of all the needed capabilities and attempt to choose these so they have an obvious and efficient translation in the object machines.

K2. The language will support the integration of separately written modules into an operational program.

Separately written modules in the form of routines and type definitions are necessary for the management of large software efforts and for effective use of libraries. The user will be able to cause anything in any accessible library to be inserted into his program. This is a requirement for separate definition but not necessarily for separate compilation. The decision as to whether separately defined program modules are to be maintained in source or object language form is a question of implementation efficiency, will be a local management option and will not be imposed by the language definition. The trade-offs involved are complicated by other requirements for type checking of parameters (see C6), for open subroutines (see J5), for efficient object representations (see J1), and for constant expression evaluation at compile time (see C4). In general, separate compilation increased the

difficulty and expense of the interface validations needed for program safety and reliability and detracts from object program efficiency by removing many of the optimizations otherwise possible at the interfaces, but at the same time it reduces the cost and complexity of compilation.

K3. A family of programming tools and aids in the form of support packages including linkers, loaders, and debugging systems will be made available with the language and its translators. There will be a consistent, easily used user interface for these tools.

No longer can a programming language be considered separately from its programming environment. The availability of programming tools which need not be developed or supported by individual projects is a major factor in the acceptability of a language. There is no need to restrict the kinds or form of support software available in the programming environment, and continued development of new tools should be encouraged and made available in a competitive market. It is, however, desirable that tools be developed in their own source language to simplify their portability and maintainability.

K4. A variety of useful options to aid generation, test, documentation, and modification of programs will be provided as support software available with the language or as translator options. As a minimum, these will include program editing, post-mortem analysis and diagnostics, program reformatting for standard identations, and cross-reference generation.

There will be special facilities to aid the generation, test, documentation, and modification of programs. The "best"

set of capabilities and their proper form is not currently known. Since nonstandard translator options and availability of nonstandard software tools and aids do not adversely affect software commonality, the language definition and standards will not dictate arbitrary choices. Instead, the development of language-associated tools and aids will be encouraged within the constraint of implementing and supporting the source language, as defined. Tools and debugging aids will be source-language oriented.

Some of the translator options which have been suggested and may be useful include the following. Code might be compiled for assertions which would give run time warnings when the value of the assertion predicate is false. It might provide run-time tracing of specified program variables. Dimensional analysis might be done on units-of-measure specifications. Special optimizations might be invoked. There might be capability for timing analysis and gathering run-time statistics. There might be translator-supplied feedback to provide management visibility regarding progress and conformity with local conventions. The user might be able to inhibit code generation. There might be facilities for compiling program patches and for controlling access to language features. The translator might provide a listing of the number of instructions generated against corresponding source inputs or an estimate of their execution times. It might provide a variety of listing options.

K5. The source language will permit inclusion of assertions, assumptions, axiomatic definitions of data types, debugging specifications, and units of measure in programs. Because many assertional methods are not yet powerful enough for practical use, nor sufficiently well developed for standardization, they will have the status of comments.

There are many opinions on the desirability, usefulness, and proper form for each of these specifications. Better program documentation is needed and specifications of these kinds may help. Specifications also introduce the possibility of automated testing, run-time verification of predicates, formal program proofs, and dimensional analysis. The language will not prohibit inclusion of these forms of specification if and when they become available for practical use in programs. Assertions, assumptions, axiomatic definitions, and units of measure in source-language programs should be enclosed in special brackets and treated as interpreted comments -- comments delimited by special-comment brackets and which may be interpreted during translation or debugging to provide units analysis, verification of assertions and assumptions, etc. -- but whose interpretation would be optional to translator implementations.

B. TRANSLATORS

L1. No implementation of the language will contain source-language features which are not defined in the language standard. Any interpretation of a language feature not explicitly permitted by the language definition will be forbidden.

This guarantees that use of programs and software subsystems will not be restricted to a particular site by virtue of using their unique version of the language. It also represents a commitment to freezing the source language, inhibiting innovations and growth in the form of the source language, and confining the base language to the current state of the art in return for stability, wider applicability of software tools, reusable software, greater software visibility, and increased payoff for tool-building efforts. It does not, however, disallow library definition optimizations which are translator-unique.

L2. Every translator for the language will implement the entire base language. There will be no subset implementations of the base language.

If individual compilers implement only a subset of the language, then there is no chance for software commonality. If a translator does not implement the entire language, it cannot give its users access to standard supported libraries or to application programs implemented on some other translator. Requiring that the full language be implemented will be expensive only if the base language is large, complex, and non-uniform. The intended source language product from this effort is a small, simple, uniform base lanauage with the specialized features, support packages, and complex features relegated to library routines not requiring direct translator support. If simple, low-cost translators are not feasible for the selected language, then the language is too large and complex to be standardized and the goal of language commonality will not be achievable.

L3. The translator will minimize compile time costs. A goal of any translator for the language will be low-cost translation, (when optimization is disabled).

Where practical and beneficial, the user will have control over the level of optimization applied to his programs. The programmer will have control over the trade-offs between compile-time and run-time costs. The desire for small, efficient translators that can be hosted by machines with limited size and capability should influence the design of the base language against inclusion of unnecessary features and towards systematic treatment of features which are included. The goal will be effective use of the available machines, both in object execution and translation, and not maximal speed of translation.

Translation costs depend not only on the compiler but the language design. Both the translator and the language design will emphasize low-cost translation, but in an environment of large and long-lived software products, this will be secondary to requirements for reliability and maintainability. Language features will be chosen to ensure that they do not impose costs for unneeded generality and that needed capabilities can be translated into efficient object representations. This means that the inherent costs of specific language features in the context of the total language must be understood by the designers, implementers, and users of the language. One consequence should be that trivial programs compile and run in trivial time. On the other hand, significant optimization is not expected from a minimal cost translation.

L4. Translators will be able to produce code for a variety of object machines. The machine-independent parts of translators might be built independently of the code generators.

There is currently no common, widely used computer in the DoD. There are at least 250 different models of commercial machines in use, along with many specialized machines. A common language must be applicable to a wide variety of models and sizes of machines. Translators might be written so they can produce object code for several machines. This reduces the proliferation of translators and makes the full power of an existing translator available at the cost of producing an additional code generator.

L5. The translator need not be able to run on all the object machines. Self-hosting is not required, but is often desirable.

The DoD operational programming environment includes many small machines which are unable to support adequately the design, documentation, test, and debugging aids necessary for the

development of timely, reliable, or efficient software. Large machine users should not be penalized for the restrictions of small machines when a common language is used. On the other hand, the size of machines which can host translators should be kept as small as possible by avoiding unnecessary generality in the language.

L6. The translator will do full syntax checking, will check all operations and parameters for type compatibility, and will verify that all language-imposed semantic restrictions on the source programs are met. It will not automatically correct errors detected at compile time.

The purpose of source language redundancy and avoidance of error-prone language features is reliability. The price is paid in programmer inconvenience in having to specify his intent in greater detail. The payoff comes when the translator checks that the source program is internally consistent and adheres to its authors' stated intentions. There is a clear trade-off between error avoidance and programming ease; surveys conducted by the Services show that the programmers as well as managers will opt for error avoidance over ease when given the choice. The same choice is dictated by the need for well-documented, maintainable software.

L7. The translator will produce compile time explanatory diagnostic error and warning messages. A suggested set of error and warning situations will be provided as part of the language definition.

The translator will attempt to provide the maximal useful feedback to its user. Diagnostic messages will not be coded, but will be explanatory and in source-language terms. Translators will continue processing and checking after errors have been found, but should be careful not to generate erroneous

messages because of translator confusion. The translator will always produce correct code; when source program errors are encountered by the translator or referenced program structures omitted, the compiler will produce code to cause a run-time exception condition upon any attempt to execute those parts of the program. Warnings will be generated when a source-language construct is exceptionally expensive to implement on the specified object machine. A suggested set of diagnostic messages, provided as part of the language definition, contributes to commonality in the implementation and use of the language. The discipline of designing diagnostic messages keyed to the design may also uncover pitfalls in the language design and thereby contribute to a more precise and better-understood language description.

L8. The characteristics of translator implementations will not be dictated by the language definition or standards.

The adoption of a common language is a commitment to the current state of the art for programming language design for some duration. It does not, however, prevent access to new software and hardware technology, new techniques, and new management strategies which do not have an impact on the source language definition. In particular, innovation should be encouraged in the development of translators for a common language, providing they implement exactly the source language as defined. Translators, like all computer programs, should be written in expectation of change.

L9. Translators for the language will be written in their own source language.

There will be at least one implementation of the translator in its own language which does all parsing and compile-time checking and produces an output suitable for easy translation

to specific object machines. If the language is well-defined and uniform in structure, a self-description will contribute to understanding of the language. The availability of the machine-independent portion of a translator will make the full power of the language available to any object machine at the cost of producing an additional code generator (whose cost may be high) and it reduces the likelihood of incompatible implementations. Translators written in their own source language are automatically available on any of their object machines, providing the object machine has sufficient resources to support a compiler.

C. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

M1. The language will be composed from features which are within the state of the art and any design or redesign which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project.

The adoption of a common language can be successful only if it makes available a modern programming language compatible with the latest software technology and with "best" current programming practice, but the design and implementation of the language should not require additional research or use of untried ideas. State of the art cannot, however, be taken to mean that a feature has been incorporated in an operational DoD language and used for an extended period, or DoD will be forever tied to the technology of FORTRAN-like languages; but there must be some assurances through analysis and use that its benefits and deficiencies are known. The larger and more complex the structure, the more analysis and use that should be required. Language design should parallel other engineering design efforts in that it is a task of consolidation

and not innovation. The language designer should be familiar with the many choices in semantic and syntactic features of language and should strive to compose the best of these into a consistent structure congruous with the needed characteristics. The language should be composed from known semantic features and familiar notations, but the use of a proven feature should not necessarily impose that notation. The language must not just be a combination of existing features which satisfy the individual requirements, but must be held together by a consistent and uniform structure which acts to minimize the number of concepts, consolidates divergent features, and simplifies the whole.

M2. The semantics of the language will be defined unambiguously and clearly. To the extent a formal definition assists in attaining these objectives, the language's semantics will be specified formally.

A complete and unambiguous definition of a common language is essential. Otherwise, each translator will resolve the ambiguities and fill in the gaps in its own unique way. There are currently a variety of methods for formal specification of programming language semantics, but it remains a major effort to produce a rigorous, formal description, and the resulting products are of questionable practical value. The real value in attempting a formal definition is that it reveals incomplete and ambiguous specifications. An attempt will be made to provide a formal definition of any language selected, but success in that effort should not be requisite to its selection. Formal specification of the language might take the form of an axiomatic definition, use of the Vienna Definition Language, or use of some other formal semantic system.

M3. The user documentation of the language will be complete and will include both a tutorial introductory description and a formal in-depth description. The language will be defined as if it were the machine-level language of an abstract digital computer.

The language should be intuitively correct and easily learned and understood by its potential users. The language definition might include an Algol-60-like description (Ref. 10) with the source language syntax given in BNF or some other easily understood metalanguage and the corresponding semantics given in English. As with the descriptions of digital computer hardware, the semantics and syntax of each feature must be defined precisely and unambiguously. The action of any legal program will be determinable from the program ard the language description alone. Any computation which can be described in the language will ultimately draw only on capabilities built into the language. No characteristics of the source language will be dependent on the idiosyncrasies of its translators.

The language documentation will include syntax, semantics, and examples of each language construct, listings of all key words and language-defined defaults. Examples shall be included to show the intended use of language features and to illustrate proper use of the language. Particularly expensive and inexpensive constructs will be pointed out. Each document will identify its purpose and prerequisites for its use.

M4. The language will be configuration-managed throughout its total life cycle and will be controlled at the DoD level to ensure that there is only one version of the source language and that all translators conform to that standard.

Without controls, a common language may become another umbrella under which new languages proliferate while retaining the common language's name. All compilers will be tested and certified for conformity to the standard specification and freedom from known errors prior to their release for use in production projects. The language manager will be on the OSD staff, but a group within the Military Departments or Agencies might act as the executive agent. A configuration control board will be instituted with user representation and chaired by a member of the OSD staff.

M5. There will be identified support agent(s) responsible for maintaining the translators and for associated design, development, debugging, and maintenance aids.

Language commonality is an essential step in achieving software commonality, but the real benefits accrue when projects and contractors can draw on existing software with assurance that it will be supported, when systems can build from off-the-shelf components, or at least with common tools, and when efforts can be expended in expanding existing capabilities instead of building from scratch. Support of common, widely used tools and aids should be provided independently of projects if common software is to be widely used. Support should be on a DoD-wide basis, with final responsibility resting with a stable group or groups of qualified in-house personnel.

M6. There will be standards and support agents for common libraries, including application-oriented libraries.

In a given application of a programming language, three levels of the system must be learned and used: the base language, the standard library definitions used in that application area, and the local application programs. Users are responsible for the local application programs and local definitions,

but not for the language and its libraries, which are used by many projects and sites. A principal user might act as agent for an entire application area.

REFERENCES

1. Space and Missile Systems Organization, AFSC, *Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s (CCIP-85)*, Vol. IV, *Technology Trends: Software*, October 1973, AD 919267L.

2. David A. Fisher, "Automatic Data Processing Costs in the Defense Department", Institute for Defense Analyses Paper P-1046, October 1974.

3. Malcolm R. Currie, Director, Defense Research and Engineering, in Memorandum to the Assistant Secretaries of the Military Departments (R&D), Subject: DoD Higher Order Programming Language, January 28, 1975.

4. Stephen A. Schuman (Ed.) *Proceedings of the International Symposium on Extensible Languages, SIGPLAN Notices*, Vol. *6*, No. 12, December 1971. Also, C. Christensen and C. J. Shaw (Ed.), *Proceedings of the Extensible Language Symposium, SIGPLAN Notices 4*, August 1969.

5. Niklaus Wirth, "An Assessment of the Programming Language PASCAL," *Proceedings of the International Conference on Reliable Software*, 21-23 April 1973, pp. 23-30.

6. Jacob Palme, "SIMULA as a Tool for Extensible Program Products", *SIGPLAN NOTICES*, Vol. *9*, No. 4, February 1974.

7. Donald E. Knuth, "Structured Programming with *go to* Statements," *ACM Computer Surveys*, Vol. *6*, No. 4, December 1974.

8. E. W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in *Structured Programming* by O-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Academic Press, 1972.

9. Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at University of California-Irvine, October 1974.

10. P. Naur (Ed.), "Revised Report on the Algorithmic Language Algol-60," *Communication of the A.C.M.* Vol. 6, No. 1, January 1963, pp. 1-17.

APPENDIX

Organizations and Individuals Contributing to the Common Language Requirements Effort

APPENDIX

Organizations and Individuals Contributing to the Common Language Requirements Effort

ARMY

U.S. Army Aviation Systems Command
St. Louis, MO

U.S. Army B.R.I.

U.S. Army Communications Command
Ft. Huachuca, AZ 85613

U.S. Army Computer Systems Command
Ft. Belvoir, VA 22060

U.S. Army Electronics Command
Ft. Monmouth, NJ 07703

Atmospheric Sciences Laboratory
White Sands Missile Range, NM 88002

Comm/Int Tech. Area EW Lab

Computer Hardware Tech Area

Electronics Tech and Devices Lab

Night Vision Laboratory
Ft. Belvoir, VA 22060

Radar Tech Area CS&TA Lab

Systems and Programming Division

Switching Tech Area

U.S. Army Force Development Command
Ft. McPherson, GA 30330

U.S. Army Intelligence Center and School
Ft. Huachuca, AZ 85613

U.S. Army Material Command
Ft. Monmouth, NJ 07703

Army Tactical Communications Systems

Army Tactical Data Systems

Navigation/Control Systems

Remotely Monitored Battlefield Sensor System

U.S. Army Mobility Equipment Research and Development Center
Ft. Belvoir, VA 22060

U.S. Army Tank-Automotive Command
Warren, MI 48090

U.S. Army Test and Evaluation Command
Aberdeen Proving Grounds, MD 21005

U.S. Army Training and Doctrine Command
Ft. Monroe, VA 23651

U.S. Army Training Support Activity
Ft. Eustis, VA 23604

U.S. Army Troop Support Command
4300 Goodfellow Blvd
St. Louis, MO 63166

U.S. Army Security Agency, Management Information Systems
Arlington Hall Station
Arlington, VA 22212

U.S. Army White Sands Missile Range
White Sands Missile Range, NM 88002

Ballistic Missile Defense Project Office
1300 Wilson Blvd
Arlington, VA 22209

Frankford Arsenal
Philadelphia, PA 19137

Harry Diamond Laboratories
2800 Powder Mill Road
Adelphi, MD 20783

Modern Army Selected Systems Test Evaluation and Review
Ft. Hood, TX 76544

Office of Chief of Engineers
Washington, DC 20314

Office of Chief of Staff
Washington, DC 20314

Office of Chief of Staff for Intelligence
Washington, DC 20310

Office of the Surgeon General
Washington, DC 20310

Picatinny Arsenal
Dover, NJ 07801

Redstone Arsenal
Redstone Arsenal, AL 35809

NAVY

Naval Air Development Center
Warminister, PA 18974

Naval Air Systems Command
Washington, DC

Naval Air Engineering Center
Lakehurst, NJ

Naval Air Test Center
Patuxent River, MD 20670

Naval Electronic Systems Test and Evaluation Detachment
Patuxent River, MD 20670

Office of the Oceanographer of the Navy
Alexandria, VA 22332

ASW Systems Project Office
National Center Bldg #1
Arlington, VA

United States Naval Academy
Annapolis, MD 21402

Naval Underwater Systems Center
New London, CT 06320

Naval Underwater Systems Center Headquarters
Newport, RI 02840

Naval Undersea Center
San Diego, CA 92132

Naval Surface Weapons Center Headquarters
White Oak, Silver Spring, MD 20910

Naval Surface Weapons Center, Dahlgren Laboratory
Dahlgren, VA

David W. Taylor Naval Ship R&D Center
Naval Ship Research and Development Center HQS.
Bethesda, MD 20034

Naval Sea Systems Command
Washington, DC 20362

Fleet Combat Direction Systems Support Activity
Virginia Beach, VA 23461

Fleet Combat Direction Systems Support Activity
San Diego, CA 92147

Naval Material Command

Naval Electronics Laboratory Center
San Diego, CA

Naval Intelligence Command

Naval Postgraduate School
Monterey, CA

Naval Research Laboratory
Washington, DC

Naval Weapons Center
China Lake, CA

AIR FORCE

Aerospace Defense Command
Ent AFB, CO 80912

Air Force Accounting and Finance Center
Denver, CO 80205

Air Force Audit Agency
Norton AFB, CA 92409

Air Force Communications Service
Richards Gebaur AFB, MO 64030

Air Force Data Automation Agency
Gunter AFB, AL 36114

Air Force Intelligence Service
Washington, DC 20330

Air Force Logistics Command
Wright-Patterson AFB, OH 45433

Air Force Military Personnel Center
Randolph AFB, TX 78148

Air Force Systems Command
Andrews AFB, Washington, DC

Aeronautical Systems Division ASD/RWSV
Wright-Patterson AFB, OH 45433

Air Force Avionics Laboratory
Wright-Patterson AFB, OH 45433

Armament Development and Test ADTC/TSX Center
Eglin AFB, FL 32542

Directorate of Computer Resource Development,
Policy & Planning
AFSC/XRF
Andrews AFB, Washington, DC

Electronic Systems Division
ESD/MCI
L. G. Hanscom AFB, MA 01730

Rome Air Development Center
RADC/ISI
Griffiss AFB, NY 13441

Space and Missile Systems Organization
SAMSO/DYVC
Los Angeles, CA 90009

Air Force Test and Evaluation Center
Kirtland AFB, NM 87115

Air Training Command
Randolph AFB, TX 78148

Air University
Maxwell AFB, AL 36112

Alaskan Air Command
APO Seattle, WA 98742

Military Airlift Command
Scott AFB, IL 62225

Strategic Air Command
Offutt AFB, NE 68113

Tactical Air Command
Langley AFB, VA 23665

United States Air Force Academy
USAF Academy, CO 80840

United States Air Force Security Service
San Antonio, TX 78243

INDUSTRY

Aerospace Corporation

Boeing Aerospace Company

Bolt, Beranek, and Newman, Inc.

Burroughs Corporation

Charles Stark Draper Laboratory, Inc.

Computer Sciences Corporation

General Electric Company

Grumman Aerospace Corporation

Hughes Aircraft Company

Intermetrics, Inc.

International Business Machines Corporation

Litton Systems, Inc.

Massachusetts Computer Associates, Inc.

McDonnell Douglas Astronautics Company

Mellanics

Research and Consulting, Inc.

Rolm Corporation

Scientific Applications, Inc.

Singer Company

Sof Tech

Sperry Univac

Systems Control, Inc.

Texas Instrument Company

TRW Systems Group

Westinghouse Defense and Electronics Systems Center

Xerox Palo Alto Research Center

OTHER ORGANIZATIONS AND INDIVIDUALS

Defense Advanced Research Projects Agency
Washington, DC

Defense Communications Agency
Washington, DC

Lawrence Livermore Laboratory
University of California
Livermore, CA

National Aeronautics And Space Administration
Washington, DC

James J. Besemer
Purdue University
West Lafayette, IN

Thomas E. Cheatham, Jr.
Harvard University
Cambridge, MA

Richard A. DeMillo
University of Wisconsin-Milwaukee
Milwaukee, WI

Edsger W. Dijkstra
Nuenen, The Netherlands

Philip H. Enslow
Georgia Institute of Technology
Atlanta, GA

David Gries
Cornell University
Ithica, NY

C.A.R. Hoare
Queen's University of Belfast
Belfast, Northern Ireland

Richard A. Karp
Stanford University
Stanford, CA

Peter T. Kirstein
University College London
London, England

Henry F. Ledgard
University of Massachusetts
Amherst, MA

Ralph L. London
Information Sciences Institute
University of California
Marina del Ray, CA

Stuart Madnick and Leonard Goodman
Sloan School, Massachusetts Institute of Technology
Cambridge, MA

John McCarthy
Artificial Intelligence Laboratory
Stanford University
Stanford, CA

Jacob Palme
Swedish National Defense Research Institute
Stockholm, Sweden

Ian C. Pyle
University of York
Heslington, York, England

Thomas A. Standish
University of California-Irvine
Irvine, CA

J.T. Webb
Royal Radar Establishment
Malvern, Great Britain

B.A. Wichmann
National Physics Laboratory
Teddington, Middlesex, United Kingdom

William A. Wulf
Carnegie Mellon University
Pittsburgh, PA

END

DATE

FILMED

9-10-76

NTIS