The ADA PROGRAMMING LANGUAGE: A TUTORIAL

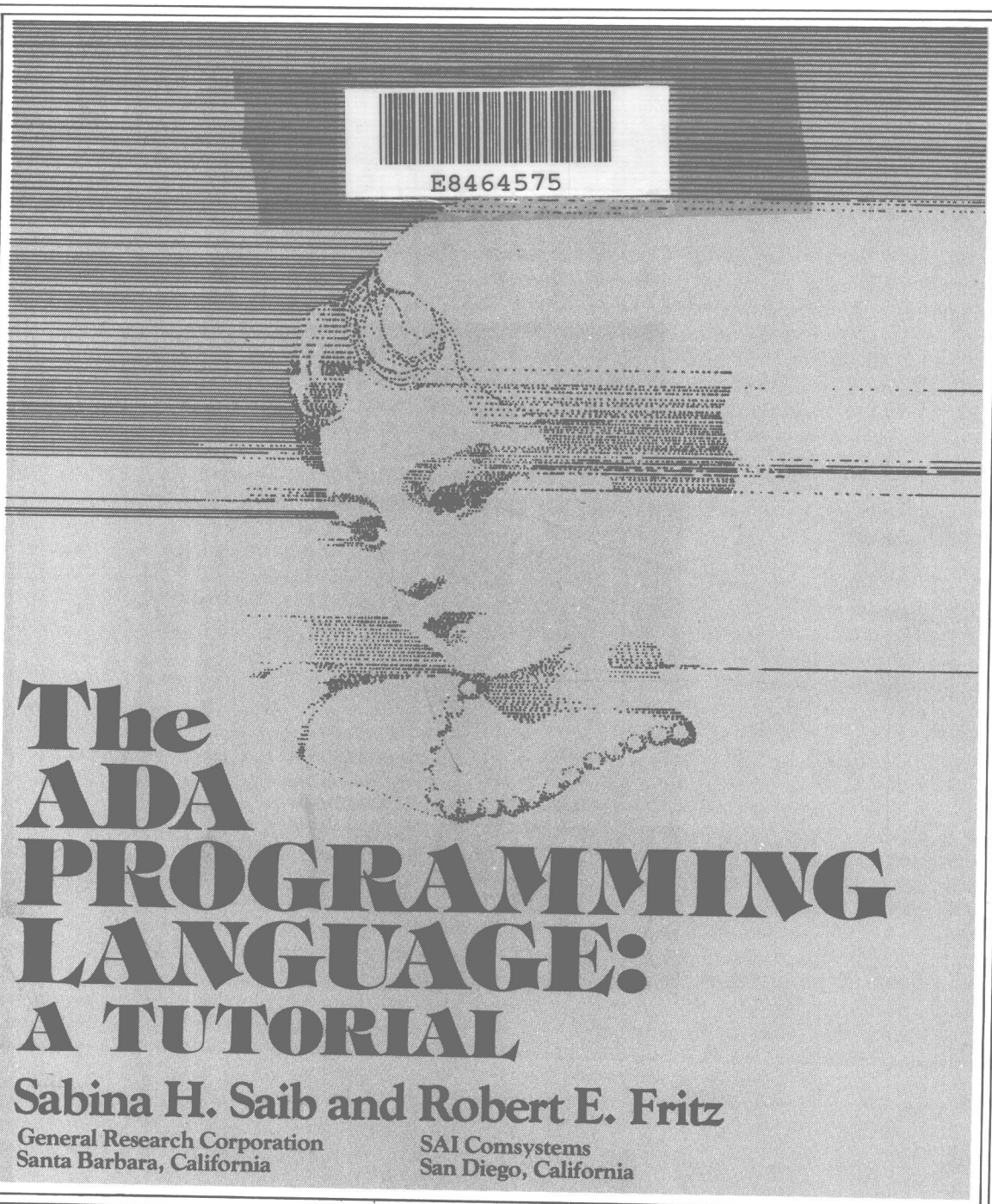
Sabina H. Saib and Robert E. Fritz

IEEE Catalog No. EHO 202-2
Library of Congress Catalog Card No. 82-84689
Order No. 456

IEEE COMPUTER SOCIETY PRESS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE COMPUTER SOCIETY



The ADA PROGRAMMING LANGUAGE: A TUTORIAL

Sabina H. Saib and Robert E. Fritz

General Research Corporation
Santa Barbara, California

SAI Comsystems
San Diego, California

IEEE Catalog No. EHO 202-2
Library of Congress Catalog Card No. 82-84689
Order No. 456

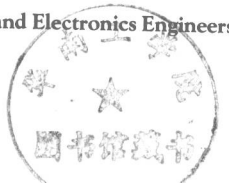
IEEE COMPUTER SOCIETY PRESS

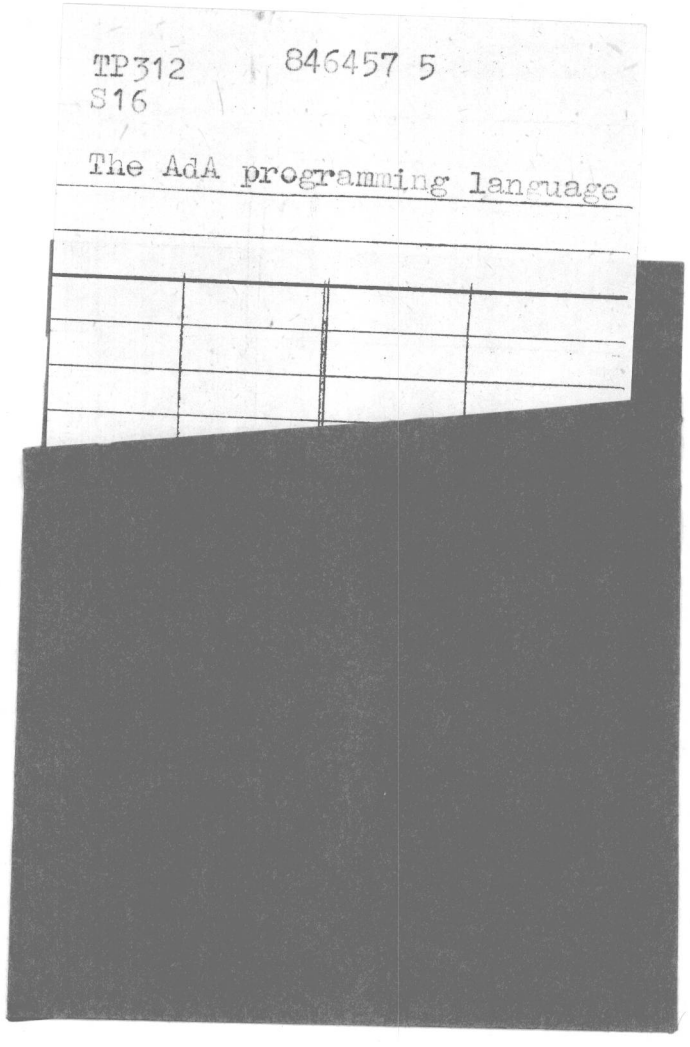
IEEE COMPUTER SOCIETY

Additional copies available from: IEEE Computer Society
PO Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lanes
Piscataway, NJ 08854

1884 1984
A CENTURY OF ELECTRICAL PROGRESS
IEEE





Dedication

To our respective spouses,
Ihsan Saib and Sharon Fritz.

Acknowledgments

Thanks go to the authors of early
papers on Ada, who have made this volume possible.
Thanks also go to Beverly Burgess for manuscript preparation, and
to Christina Taylor for her careful editing.

Foreword

This Ada tutorial brings together many of the major early papers on the Ada* programming language and its environment. It also contains a number of tutorial papers that describe several of the most important aspects of the language, notably those that apply to real-time embedded systems. It is this area of computer applications that has suffered from a lack of a high-level language; and this is just the area Ada was designed to address.

We have gathered various viewpoints on Ada and its uses, including those of some of its harshest critics. The volume also contains papers that had significant influence on the designers of Ada, a comprehensive bibliography, and a glossary of Ada terminology.

Most of the papers in this collection use examples based on the July 1980 Language Reference Manual. With a few exceptions, the papers will remain valid for the 1982 Language Reference Manual.

Sabina H. Saib
Robert E. Fritz

August 1982

*Ada is a trademark of the US Department of Defense (Ada Joint Program Office).

TABLE OF CONTENTS

PART IV: ADA DESIGN METHODOLOGY AND LANGUAGES

PART V: ADA OVERVIEW

PART VI: REAL-TIME PROGRAMMING

Part I:
The History and Current Status of Ada

As of 1982, Ada is still in the early stages of its evolution. The design is substantially fixed, implementations are progressing, and applications are beginning to appear. Numerous studies have sought the most effective methods for using Ada in all kinds of applications. Ada is ready to bloom, awaiting only the availability of complete, production-quality compilers.

Milestones

There have been three major milestones in the design of the Ada language:

- the original features of the language (June 1979);
- the revised Ada of July 1980;
- and the clarifications and small revisions of late 1982, which arose from the ANSI standardization process.

Standards. The July 1980 revisions were substantial, resulting from nearly a year of review and comment by thousands of people in industry, academia, and government. When the July 1980 version was submitted for ANSI standardization, many ANSI canvassees found ambiguous or inadequately explained areas in the language. These problems must be answered before the language can be an accepted standard, and this will require a revision of the language reference manual. When the revisions are complete, a new standardization canvas will be made and Ada will become an ANSI standard.

Considerable foreign interest in Ada has prompted submission of the language design for International Standards Organization approval. Though the ISO standard will be the same as ANSI, the ISO process will probably require a formal definition and will therefore not be complete until 1984 or 1985.

Enforcement. Standards on implementations of the language are enforced through the Ada Compiler Validation Capability, or ACVC, and through Ada's trademarked name. Through the Ada Joint Program Office, or AJPO, the US Department of Defense has registered Ada as a trademark. Only compilers that pass the ACVC tests will be certified as true Ada and allowed to use the name, and only officially certified Ada systems will be allowed on DoD projects. The goal of ACVC certification is to prohibit subsets and supersets of the language, which cause problems in transporting programs from one system to another.

Implementation

Both DoD and private enterprise are funding implementations.

DoD-funded implementations. The DoD is funding two major Ada implementations: the Army Ada Language System, or ALS, and the Air Force Ada Integrated Environment, or AIE. A preliminary version of the ALS, written in Pascal, is scheduled for completion in late 1982. The final ALS—written in Ada—is due in December 1983. The AIE should be complete in 1984. Though the AIE and ALS represent different approaches to organizing Ada's supporting environment, both will have certified production compilers.

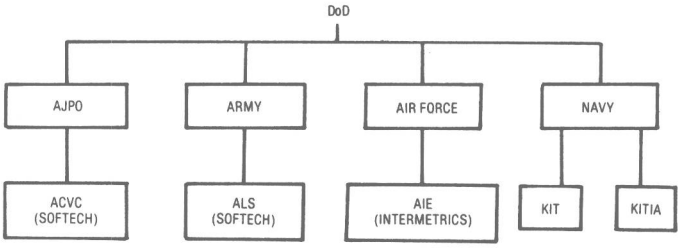


Figure I-1. Major Dod Ada projects.

The major DoD-sponsored projects, shown in Figure I-1, are complemented by an effort sponsored by the Navy. The KAPSE Interface Team, or KIT, and the KAPSE Interface Team for Industry and Academia—KITIA—seek to ensure interoperability and transportability of Ada tools among Ada environments, including AIE and ALS.

Privately Funded implementations. Private enterprise has been quick to recognize the value of Ada and its importance in the DoD. Several implementation efforts are underway; a remarkable number are based on microprocessors.

Originally, Ada compilers were considered beyond the capability of microprocessor-based systems, but

Table I-1.
Major commercial Ada compiler efforts.

DEVELOPER	PROJECT	COMPLETION TARGET
BELL LABORATORIES	VAX/UNIX* COMPILER	1982
BURROUGHS	GENERATE DIANA FROM ADA	1982
CONTROL DATA CORP.	CYBER-170/NOS COMPILER AND PARTIAL ENVIRONMENT	1985
DIGITAL EQUIPMENT CORP.	VAX/VMS COMPILER	1983
INTEL	iAPX 432 CROSS COMPILER OPERATING ON VAX/VMS	1981
TELESOFT	MC-68000, 8086, VAX, IBM 370 COMPILERS	1982
WESTERN DIGITAL	SUPERMICRO 1600	1982
ZILOG	Z-8000 COMPILER	1983

*Unix is a registered trademark of Bell Laboratories.

Intel, TeleSoft, and Western Digital, among others, are developing just such systems. The Intel compiler system uses the iAPX 432, whose hardware closely supports Ada. Telesoft is building compilers for systems based on the Motorola MC68000 and the Intel 8086 and 8088; the Telesoft Ada system is even available for the IBM Personal Computer. Western Digital is producing Ada for its series of microprocessor-based systems.

Information on implementations of Ada by manufacturers of minicomputers and mainframes is sparse, since most large manufacturers maintain some security over the status of such projects to maintain a competitive edge. But the presence of manufacturers' representatives at Ada-related meetings and conferences suggests that major implementation efforts might be underway.

Table I-1 shows some important commercial projects and milestones for Ada compilers. The table gives only an overall view of where Ada is going; it does not necessarily reflect specific delivery dates or availability.

Implementation principles. Ada compilers are being implemented according to modern principles of compiler construction, which organize compiler activities into distinct stages. The grammar of Ada can be made LALR(1), so syntax analysis can be done according to formal, but easily implemented, procedures.

Semantic analysis results are currently in the form of the Diana intermediate language. Use of an intermediate language in the compiler makes rehosting the compiler easier, since the syntax and semantic analysis stages are free of host-specific information. Code generation for the host machine is, by necessity, host-dependent; therefore, it uses the intermediate language to produce executable instructions.

Diana is based on two earlier Ada intermediate languages, TCOL Ada (by Carnegie-Mellon University) and Aida (by the University of Karlsruhe, Germany). Diana incorporates features of both, along with some new ones.

Growing support

Support for Ada continues to grow. Within the DoD, support has become policy as the military services have issued memoranda on Ada use schedules and plans. Requests for proposals that require Ada in some way on DoD projects are now appearing.

Outside the DoD, organized support, in the form of the ACM Sigplan AdaTEC group, continues to grow rapidly. This is an active national organization that meets several times each year and now has several local chapters. Many commercial Ada courses and textbooks have appeared and done well.

The future of Ada is bright. Technical issues are being resolved, implementations are underway, approaches for applying Ada are being developed, and official and unofficial policy and support are growing. The continued success of Ada depends on continued official support, the quality of initial applications, and the usefulness of the technical features of Ada.

Problem areas

Ada has its faults, problem areas, and critics. Many of its supposed language faults are matters of opinion—one person's fault is another person's feature. Problem areas generally result from inadequate specification or explanation in the original design. These are being resolved, though with considerable effort. Meaningful criticisms center on the reliability and efficiency of the language and its compiler; they cannot be answered without evidence based on substantial use of Ada. Other criticisms, based on emotional or personality reactions, deserve no response.

Issues facing Ada

Ada is a developing, high-visibility program within DoD. As such, it faces several technical and political questions. While it is unrealistic to expect the chosen answers to these questions to satisfy everyone, one can expect reasonable compromises to be acceptable to most Ada users. A goal of the program is continued responsibility to all Ada users. This goal is being reached through cooperation with such interest groups as ACM Sigplan AdaTEC and the JOVIAL/Ada Users Group. As experience further defines the needs of the Ada community, one would expect changes, within certain limits, in official Ada program policy.

Definition and standardization. Definition and standardization of the language are immediate issues. In 1981, the Language Reference Manual, or LRM, was submitted to ANSI for consideration as an ANSI standard. Part of the ANSI process is a canvass of members of the appropriate technical community on completeness, accuracy, and acceptability.

In the preliminary canvass on Ada, the LRM was conditionally accepted as a standard definition of the manual. Before the LRM could be resubmitted to ANSI canvassees, the questions and objections raised in the preliminary canvass had to be answered. Many of its discrepancies were due to technical deficiencies in the manual; several sections provided only a sketchy description of language features. These areas were amended to resolve issues of semantics.

Ambiguity, paradox. The original language description did not discuss interaction of some parts of Ada. Review by many thousands of people revealed some ambiguities and paradoxes, which had to be resolved. While this required detailed changes in several areas, it did not alter Ada's general character or structure. All changes were finally approved by Jean Ichbiah, who headed the original design team. This process has not been without controversy.

Refinement ended in July 1982. The July 1982 version of the Ada Language Reference Manual, available from AdaTEC, is the current standard. LRMs dated July or November 1980 or earlier are obsolete and substantially different. The 1982 Ada LRM is being submitted to the ANSI canvassees for approval by the fall of 1982.

Standardization within DoD. Ada is already standardized within DoD as MIL-STD-1815. The defining document is the July 1982 Ada LRM. Currently, Ada is listed on DoD Instruction 5000.31, the Interim List of Approved Languages. This instruction defines which languages may be used by contractors building embedded systems for the services. Eventually, this list will be pared to Ada alone, but it currently allows JOVIAL for Air Force projects and CMS-2 for Navy projects. The Army is currently specifying Ada for new systems.

ISO standardization. International standardization of Ada will be done by the International Standards Organization. The ISO process requires development of a formal semantic description of Ada, which is not yet available for the 1982 version. The ANSI standard will serve as the interim ISO standard.

Ada subsets

Standardization has focused on the full language, but there are several proponents of Ada subsets. They wish to see a standard subset defined.

Thus far, formally defined subsets are not allowed by the policies of the Ada Joint Program Office. Implementers have been given a grace period in which to call an incomplete implementation "Ada" and offer it as a product. These sources must also declare that they intend to complete their compiler system and submit it for validation within a reasonable time. Trademark enforcement action by AJPO is expected to increase in 1983 as more Ada systems, a standard, and the Ada Compiler Validation Capability become available.

Subsetting has been suggested to aid teaching, programming, efficiency, and to allow use on small systems. AJPO has established a firm no-subset policy, but this has done little to quiet the debate. AJPO wishes to force implementations of Ada to be complete and to establish the standard as an existing tool. Perhaps this policy will change once several validated Ada compilers are available. But it is more likely to remain in force, since many of the pro-subset arguments are losing impact as more is learned about implementing and using Ada.

Subsets for teaching Ada. Subsets have been successful in teaching aids for many languages, particularly such large languages as PL/I. Typically, a simple subset that contains basic types and control structures is chosen from the whole language and used to teach the fundamentals of programming. When the student has mastered the rudiments, other features of the language are introduced, until the entire language or a major part of it is learned.

Ada could be taught with this layered approach, but the kernel and subsequent layers must be carefully chosen. Effective use of Ada requires understanding of certain features, such as packages and extensive typing. One might be tempted, upon cursory examination, to leave these to the later lessons. Since, however, program structure is as much a part of Ada programs as algorithm design, these features should not be treated as afterthoughts.

An initial teaching subset of Ada might include the following topics, starting at the primitive level and working toward the macroscopic features. (The following sequence is not necessarily the most effective way to teach the language.)

(1) *Language basics:* name conventions, operator symbols, reserved words, program parts

(2) *Types:* predefined types, user-defined types, basic types, numeric and string types, derived types and subtypes, enumeration types, declaring variables and constants, initializing variables and constants

(3) *Expressions and statements:* using assignment, if, case, loops, forming arithmetic and logical expressions

(4) *Simple composite types:* arrays and records, constrained and unconstrained arrays, records with no discriminant parts

(5) *subprograms:* functions and procedures, parameters, named and positional parameters, scope of variables, subprograms and system structure.

(6) *Packages:* data packages; combination packages of types, variables, and subprograms; packages and system structure; using packages for encapsulation and information hiding, private types

These topics would serve as primitive Ada, the first step in learning the complete language. Topics to be introduced later include the following:

(1) *Advanced types:* discriminant records, access types, overloading values and fields in enumeration types and records, specification types, limited private types

(2) *Subprograms:* default parameters, recursion, overloading operators and subprograms

(3) *Tasks:* rendezvous, instantiation, activation, termination, protected regions, avoiding deadlock, tasks in program structure, parallelism

(4) *Generics:* types, subprograms, generics and overloading, use in system design

Of course, these topics only suggest the use of progressive subsets of Ada for teaching purposes and do not constitute a complete syllabus. The above organization adheres to traditional methods of teaching programming languages. Early results indicate that a strictly traditional approach may not provide a full appreciation of Ada's software engineering and systems programming features.

Subsets for programming efficiency. There are two approaches to subsetting for programming efficiency; one is acceptable to AJPO, but the other is contrary to its policy. The latter eliminates certain features of the language from the compiler. Tasking and generics are frequently mentioned potential victims, along with some of the more complex types.

Tasking is a likely candidate because of fears that the time and general system overhead required to switch one task to another on single-processor systems are too great. In multiprocessor systems, the complaint is that interprocessor communciation overhead is too great or that the timing is not precise enough in some situations. Also, generics are said to be difficult to implement. This is conjecture at this point, albeit conjecture based on the experience of similar structures in other languages. There is probably not enough evidence to justify a subset that lacks these features.

Other suggested subsets would retain tasks or generics, but reduce their complexity by eliminating some of their aspects. For example, guarded tasks might be removed because of the overhead in checking the guard as it comes into the entry. Again, it is not clear that surgery of any kind on the language will yield a significant advantage. Any subset that eliminates any feature of standard Ada is contrary to the policy of AJPO and of questionable gain at this time.

An AJPO-acceptable approach. An approach to subsetting Ada that is compatible with AJPO policy—though not, perhaps, with the spirit of the Ada program—is the imposition of programming style guides or standards. In this approach, a full Ada compiler processes the source code and, perhaps, other tools verify that the standards are observed. In other words, restrictions are imposed by fiat rather than by the compiler. The use of tasks might be restricted to simple cases, or the use of exotic types might be restricted. Access types and recursion could be avoided in some embedded applications, to solve the problem of garbage collection or never-ending recursive calls that occupy all available memory. For critical applications, this might be acceptable. But artificial constraints on the Ada style could prevent creation of a more elegant solution, one that might save more resources than could the restrictions.

Pare it down for micros? Some claim that in order to take advantage of the inexpensive and abundant microprocessor, Ada must be pared down. A smaller Ada would allow development on a microprocessor and efficient operation and lower overhead on a microprocessor system. This might be true for the eight-bit microprocessor, because most are limited to a 64-K byte address space. Along with slow I/O devices, the 64-K memory space probably restricts the size of the compiler to a subset of the language. It is doubtful that full Ada can be implemented on any of the currently popular eight-bit microprocessors (Z-80, 8080, 6502, 6809, etc.).

The inability to develop Ada programs on eight-bit systems should not be particularly disconcerting, because 16-bit and even a few 32-bit microprocessors are available. Most of these allow a 20-bit or larger address and have a one-megabyte address space. This is certainly adequate for a full Ada compiler and runtime support system. Evidence of the adequacy of such micros is the development of a commercial Ada compiler by TeleSoft done exclusively on a Motorola MC68000 16-bit microprocessor system. This system was transported to the Intel iAPX86 (8086/8088) and then to the DEC VAX-11/780 and the IBM 370.

The economy of the 16-bit microprocessor can be startling. Recent benchmarks have shown that the 68000 can outperform the VAX in many kinds of calculations. With the 8087 math coprocessor, the 8086 is in the same league—and faster versions of the 8086 are being introduced.

A typical VAX installation costs $500,000 or more, while a 68000 system is about $20,000. MC68000 systems have been included in some "personal computers" at a much lower cost. The 8088 is available in several personal computers. Among them is the IBM Personal Computer, which has a spot for the 8087 math coprocessor and costs considerably less than $10,000. This cost/performance ratio is unequaled by eight-bit microprocessors or minicomputers.

Thus, it is difficult to make a case for reducing Ada to fit microprocessors. The current 16-bit generation has more than adequate performance to host the Ada compiler and development system or to serve as the target system of operational software. As a target, an eight-bit microprocessor with 64-K memory can probably support most Ada applications. A microprocessor subset of Ada is one of the least justified of all proposed subsets.

Ada on small commercial systems

The availability of Ada on microprocessor sytems gives cause to consider the impact of Ada on the small business and personal-computer-system market. Most projections show explosive growth for both in the next decade.

Most business software for small systems is written in compiled Basic, interpreted Basic, Cobol, Pascal, or assembly language. Most software for home systems is in interpreted Basic or assembly language, with some Pascal and other languages available. In both markets, many programmers are self-taught, usually in Basic, and are often quite arrogant of the supposed power of their language. The techniques and details of software engineering of solutions are unknown to many of these programmers; their approach is to begin coding and

solve, with a profusion of code, all problems as they appear. Pascal programmers are generally better, but probably because the language makes program organization easier. Ada allows even more organization, at the expense of a more complex, harder-to-learn language that requires a more structured implementation approach.

For the amateur programmer, Pascal is probably the limit of self-taught language. Ada is somewhat more difficult to learn, so its impact on the amateur coder is uncertain.

Related to this problem is the plethora of high school, junior high, and even elementary students with access to computers. Here again, the language is generally Basic. They learn coding, but not software engineering of programs. Unless a precollege software engineering/computer science curriculum is developed, colleges will soon face the task of retraining students who possess bad habits learned and practiced from childhood. The availability of computers in secondary schools might prove less than beneficial if it results in undisciplined coders.

Ada will have surprising impact on systems for small businesses. These systems are often developed by professional programmers with a desire for independence and are generally of professional caliber. Tools for developing software on small systems are barely up to the task, so remarkable skill and ingenuity have been used to work around system-level shortcomings.

Advantages of Ada. Ada offers several advantages to developers of small systems, particularly the ability to easily enhance a system by adding a module with additional capabilities. Because of the package structure, a software system can be designed through a common database described in a single package that is accessible to several service components. These components can be assembled to suit the needs of the user, who pays only for the software desired. As user needs grow, the system can be enhanced by the addition of other service modules.

Such modular growth is a familiar promise, but all too often an empty one. With most current systems and languages, additional software generally means a not-so-minor revision of the existing system. Ada offers the software engineer a legitimate tool with which to effect an expandable software system.

Ada is a craftman's toolbox, not a hacker's sledge and mallet. Well-engineered systems require discipline, planning, a structured approach to systems development, and knowledge of the fundamental techniques of software engineering: data abstraction, modularity, encapsulation, information hiding, etc. As the small-systems market expands, the software designers who prepare for the long term with a flexible, portable sytem that is easily transferred from host to host will prosper. As more people learn the techniques of software engineering, Ada will become the most attractive language for development of software for long-life-cycle systems that are to be supported (maintained) for many years.

Highly cost-effective 16-bit microprocessors and other hardware, the need for powerful small-system development tools, and the expansion of application areas from simple accounting systems into real-time (or near real-time) business support and process control systems will combine to make Ada the likely small-system leader in the next decade. Surprisingly, the first delivered Ada software was for an accounting and inventory system that runs on a Motorola MC68000 system. About half of the Ada systems sold by TeleSoft have been to suppliers of small business systems with no DoD contracts.

Transition to Ada

Ada may create some of its own market, but most of its use will be by programmers with experience in other languages.

Transition to Ada requires two quite different, but related transfers: that of programmer skills in other languages and that of programs in other languages. Neither is trivial. Programmer training is required because of the new features and combinations of features available in Ada and program conversion requires more than mechanical translation. Both can be cost effective in some situations.

Programmer conversion. Programmer conversion to Ada is not as easy as handing the programmer a reference manual the day a project begins. Learning Ada in this manner guarantees a budget overrun and a late project that does not meet its specifications.

It has become an aphorism that bad programs can be written in Ada, but a programmer trained in both modern software engineering principles and ways to apply these rules to Ada will find it difficult to write really horrible programs. Some software engineering principles are familiar to the experienced programmer, since they are simply abstracted common sense. Because of language limitations, others might be unfamiliar. For example, the idea of data abstraction through user-defined types might be unfamiliar to a Fortran programmer accustomed to using those few types predefined in his language.

Neither is Pascal experience an entirely satisfactory education for Ada. Though its algorithmic features resemble those of Ada, Ada has additions that make it subtly different and quite a bit more powerful. Ada's system programming features, for example, are unmatched by Pascal.

Pascal programmers are deceived by the apparent ease with which they produce working Ada programs. On a small scale, this is no problem. On the large scale, however, the Pascal mindset can seriously degrade the performance of the system.

Ada must be learned and used as a distinct language tool; it differs sufficiently from any other to warrant a training program devoted to its proper use.

User implementation issues

From the user's standpoint, Ada must satisfy certain performance issues. Among them are fast task-switch time; disabling of runtime checks; effective garbage collection (to make recursion and access types thinkable

for real-time applications); and effective optimization techniques for compiled code. Of course, users will be affected by the implementation of Ada in other ways; these are just a few of the obvious ones.

Overview: The papers in Part I

The first three papers in Part I, by Whitaker, Carlson, and Fisher, present the ideas behind the development of Ada and the early steps and plans for its implementation. These authors, responsible for managing the development of Ada from its earliest days, were DoD employees.

"The U.S. Department of Defense Common High Order Language Effort" by W. A. Whitaker, presents much of the early background of the language. Its development was motivated primarily by a desire to avoid the costs of developing and maintaining software in a variety of languages. The success of Cobol served as an inspiration for the design of a language that would meet the needs of most real-time applications. The goals for the new language were to reduce the life-cycle cost of software; ease transportability of software across projects and among computers; and assist maintenance, reliability, and readability. It was to compare in efficiency with the best possible machine-language coding.

"DoD's Common Programming Language Effort" by David Fisher, details the evaluation of existing languages against a set of requirements for a high-order language that would satisfy a large number of applications. He also lists the steps in developing the requirements for Ada. No existing language met all the requirements, but Algol-68, Pascal, and PL/I could have been modified to meet them. Instead, a design competition was established to produce a new language, which was eventually named Ada. When Fisher and Whitaker wrote their papers, the language had not yet been named.

"Ada: A Promising Beginning" is by William Carlson, who worked closely with the other two authors at the Defense Advanced Research Projects Agency. Carlson wrote it after the Ada design had been selected. He lists Ada's advanced features and some managerial features of a common language. For example, he notes that Ada will be taught to a large number of programmers and will be part of most modern software engineering curricula. Carlson also lists the problems that Ada was designed to solve and the approach taken to solving them. He argues for freezing the language pending experience in its use and describes the use of a validation test set to control the language.

"From Pascal to Pebbleman . . . and Beyond" by Robert Glass, gives some history behind the name "Ada" and some background for the next stage in the development of the Ada environment. He says that the Ada environment will contain a standard set of software tools: an editor, compiler, linker, debugger, and a configuration manager, which will assist in the development of Ada programs. He further notes that the Ada environment will be written in Ada and will be easily transported to a variety of machine configurations. The paper relates this national effort to similar European and Japanese projects.

"What is Ada" by Ronald Brender and Isaac Nassi, is a brief overview of the most important features of the Ada programming language. Of particular importance are the features supporting modularity and real-time programming.

Ada packages are probably Ada's main contribution to programming language design. Brender and Nassi believed that packages would aid in the grouping of software subprograms into modules, which could be reused in many projects without reprogramming. Packages also support abstraction or information hiding, which can help in program maintenance. Another major contribution of Ada is the mechanism for multiprocessing. Ada supports multiple tasks, protected communciation among tasks, priorities for tasks, and scheduling of tasks and interrupts.

THE U.S. DEPARTMENT OF DEFENSE COMMON HIGH ORDER LANGUAGE EFFORT

William A. Whitaker, Lt.Col., USAF
Defense Advanced Research Projects Agency
1400 Wilson Blvd., Arlington, Va. 22209, USA

The United States Department of Defense (DoD) spends about three billion dollars a year on computer software. This includes the design, development, acquisition, management, and operational support and maintenance of such software. Only a small fraction of this effort is involved with the accounting, inventory, payrolling, and financial management functions which are defined by the Federal Government as Automatic Data Processing, those functions that have their exact analogy in the commercial sector and share a common technology, both hardware and software. A much larger fraction of the DoD's computer investment is in computer resources which are embedded in, and procured as part of, major weapons systems, communications systems, command and control systems, etc. In this environment the DoD finds itself spending an even larger share of its systems resources on software. As a result, this area is receiving increasing attention from the highest levels of management. A number of technical and managerial initiatives have been called out to both reduce the cost and improve the quality of Defense systems software. A management plan has been formulated in this area and initial guidance is provided by DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems.

In the area of software we may have, at the present time, more flexibility and a greater influence on the technology than with hardware. Some years ago, the DoD was a major innovator and consumer of the most sophisticated possible computer hardware. It now represents only a small fraction of the total commercial market. In software, that unique position still maintains. A significant fraction of the total software industry is devoted to DoD related programs and that is true in even larger proportion for the more advanced and demanding systems. Thus, there is both an opportunity and a responsibility in the software arena which is past for hardware.

One specific initiative which has been called out by DoD Directive 5000.29 is the use of high order languages (HOL) in systems development. The advantages are well known and in many communities, for instance, the COBOL financial management community or the FORTRAN scientific computational community, these advantages are so persuasive that there has been essentially no alternative to the use of these common languages for more than a decade. The obvious advantages include ease of writing of programs, self-documentation, ease of maintenance, ease of modification, transportability of programs, simplification of training, etc.

Reprinted from *ACM Sigplan Notices*, February 1978, pp. 19-29.

It is surprising that a general consensus has not mandated a common high order language for embedded systems long since. There are, however, a number of managerial technical constraints that have acted against this in the past. For most Defense systems applications, very severe timing and memory considerations have been prominent in the past, often governed by real time interaction with the exterior environment. Because of these constraints, and restrictions in developmental cost and time scale, many systems have opted for assembly language programming. This decision is often substantially influenced by past experience with poor quality compilers and the fact that the assembler comes with the machine, while the compiler and its tools usually must be developed after the project has begun. The advantages of high order languages, however, are compelling and many more recent systems developments have turned to HOLs. Because of limitations of available high order languages, the programs generated most often include very large portions done in assembly code and linked to an HOL structure, negating many of the expected advantages.

Further, many systems have found it convenient to produce their own high order language or some perhaps incompatible dialect of an existing one. Since there is no general facility for control of existing languages, each systems office has had to do the configuration control on their language and compilers and continue to maintain such on their particular dialect through the entire maintenance phase of the system, which may be very long lived. This has had the effect of practically reducing the contractual flexibility of the government and restricting competition in maintenance and further development. This lack of commonality negates many advantages of high order languages including transportability, sharing of tools, the development of very powerful tools of high efficiency and, in fact, not only raises the total cost of existing tools, but in some cases essentially prices them out of the market. Many development projects are very poorly supported and forced to live with a technology which is far below the state-of-the-art.

By the early 1970's each of the military departments had underway studies or actual language designs which were expected to lead to common languages for large portions of those departments, in January 1975 the Director of Defense Research and Engineering set up a Defense-wide program with the goal of a single common military computer programming language for embedded systems. The intent was to have a real time language to supersede those numerous ones in existence while maintaining the standards of FORTRAN and COBOL, the success of which standards had provided impetus to this consolidation program. Further, to assure non-proliferation during the duration of this effort all other implementations of new high order programming languages for R&D programs were halted. A High Order Language Working Group (HOLWG) with representatives from DoD and the Military Services was established as the agent for this effort.

Briefly, the logic of this initiative is as follows:

- The use of a high order language reduces programming costs, increases the readability of programs, the ease of their modification, facilitates maintenance, etc. and generally addresses many of the problems of life cycle program costs.

- A modern powerful high order language performs these tasks better and, in addition, may be designed to serve also in the specification phase and provide facilities for automatic tests and program verification. A modern language is required if real time, parallel processing, and input/output portions of the program are to be expressed in high order language rather than assembly language inserts which destroy most of the readability and transportability advantages of using an HOL. A modern language may also provide better error checking, more reliable programs, and the capability for more efficient compilers.

- Many of the advantages of a high order language can only be realized through computer tools. A total programming environment for the language includes not just compilers and debugging aids but text editors and interactive programming assistance, automatic testing facilities and proofs of correctness, extensive module libraries, and even semi-automatic programming from specifications. Universal use of those tools which are available today would significantly reduce the present cost of software. Development of more powerful tools holds even greater promise. Unfortunately, the average programmer's tool box is rather bare. Because of the difficulty of preparing these tools for each new language and machine and operating system, and the time involved, only the very largest projects have been able to assemble even a representative set. While in many cases development of tools can be shown to be desirable in the long run, day to day pressures usually prevail. There is almost never time to do it right. The use of a common high order language across many projects, controlled at some central facility, would allow the sharing of resources in order to make available the powerful tools which no single project could generate. It would even make those previously generated tools available at the beginning of a project, reducing start up time.

- Reducing the number of languages supported to a minimal number, therefore, provides the greatest economic benefit. There are, of course, costs associated with supporting any particular project and general costs of supporting the language. For a sufficiently large number of users, presumably the basic cost would be proportionally less. Perhaps 200 active projects contributing to a single support facility may not be proportionally much cheaper than two facilities each supporting 100 projects, although the absolute saving would be significant.

- There are, however, unique advantages to having a single military computer language. With a single language, one could reasonably expect new computers proposed for a project to be supplied by the manufacturer with a compiler. This is, in fact, the experience of the British with their common language effort. If there were five or ten common languages, that is not a reasonable expectation. In fact, if there were a single common language, its use in DoD and the provision of tools by the DoD would make it a popular candidate for