N86-24533

1985

NASA/ASEE SUMMER FACULTY RESEARCH FELLOWSHIP PROGRAM

MARSHALL SPACE FLIGHT CENTER
THE UNIVERSITY OF ALABAMA

ANALYSIS OF HIGH-ORDER LANGUAGES FOR USE ON SPACE STATION APPLICATION SOFTWARE

Prepared by:	Arthur Knoebel
Academic Rank:	Professor
University and Department:	New Mexico State University Department of Mathematical Sciences
NASA/MSFC: Division: Branch:	 Software and Data Management Systems Software
MSFC Counterpart:	John W. Wolfsberger

Date:	August 15, 1985
Contract No.:	NGT 01-008-21 (The University of Alabama)

Analysis of High-order Languages for Use on Space Station Application Software

by

Arthur Knoebel
Professor of Mathematical Sciences
New Mexico State University
Las Cruces, New Mexico

ABSTRACT

Considered in this study is the general and not easily resolved problem of how to choose the right programming language for a particular task. This is specialized to the question of which versions of what languages should be chosen for the multitude of tasks that the Marshall Space Flight Center will be responsible for in the Space Station. Four criteria are presented: theoretical considerations, quantitative matrices, qualitative benchmarks, and the monitoring of programmers. The report concludes with specific recommendations for future studies to resolve these questions for the Space Station.

ACKNOWLEDGEMENTS

John Wolfsberger, with his incredible energy, is creating a language laboratory whose rare climate is extraordinarily stimulating to work in. Bob Stevens, Dave Aichele and Gabe Wallace have been very supportive and sensitive to the needs of a summer fellow. Gerry Karr, with his dry sense of humor, has administered the summer program with aplomb and grace. Jim Dozier and Leroy Osborn planned and executed an informative round of talks and tours. Dina Conrad, with much patience, has thoughtfully tended to the day-to-day needs of the summer faculty. Jimmy Watkins and Ester Gill, my office mates, have been quite helpful in breaking down the resistance of recalcitrant computer programs. All of these deserve a round of applause and a bouquet of flowers for making my summer a profitable busman's holiday away from academia.

TABLE OF CONTENTS

ORIGINAL PAGE IS OF POOR QUALITY

INTRODUCTION

Problem As one of the future major projects of NASA with a long lead time, the Space Station is a good place to reconsider software and hardware decisions made in the past. This in fact is being done: it appears that Ada will be chosen as the principal or preferred programming language for the Space Station. Given that this happens, we still have some significant questions to answer. What should the backup language be, in case the current doubts about Ada materialize as experience with this new, large and untried language is gained. What about special purposes? Certainly we should allow other languages specially designed for such areas as robotics and artificial intelligence. And if we allow alternates, then we must select the right dialect. Clearly there is much to resolve in the language arena.

History of Languages. A brief history of programming must start with assempbly languages forty years ago. To this FORTRAN added algebraic expressions. Control abstraction came with Algol. Pascal is notable for a variety of data types and the ability to check their conformity at compile time. Pascal's biggest drawback, the inability to compile parts of a program separately, was taken care of in a variety of languages.

At Marshall Space Flight Center, the four most popular languges by far have been assembly , FORTRAN, HAL/S and C. Assembly has been essential from the first for real-time applications, where until recently there have been no high-order languages with adequate input-output facilities for communicating with sensors and servos. FORTRAN, the perennial bellwhether, is now used principally for calculations in scientific research and engineering design . The space shuttle gave birth to HAL/S, and no doubt HAL/S will die with it. C is the least used at MSFC.

Language Choice. Does it make a difference what language is chosen? Yes, it certainly does. For surprisingly not all so-called high order languages are even equivalent computationally to a primitive Turing machine; we discuss this point in more detail in the body of this report. For Turing-machine equivalent languages, the selection of primitive operations, data types and control structures can make a big difference in the ease of programming and the avoidance of errors. Also, a variety of features may provide for more extensive error checking at compile time.

The need for an intelligent decision on a programming language comes about for a variety of reasons. The kind of task is all important -- is it numerical or symbolic, real-time or batch, for research or production? How big is the task -- is it for one programmer over a month or two, or at the other extreme, for more that a hundred programmers over a several years? How many programmers -- one programmer can be somewhat sloppy and informal about direction and management, in a multi-programming some kind of configuration controls are essential. and modularity of the language can help here. The complexity of the task makes a difference too -- here we don't mean size; rather are the parts of the problem to be programmed similar in style or are there many different concepts to be integrated?

Objectives. The original objectives were five in number:

1) Review Space Station mission requirements to establish variant needs;

2) Identify logical language candidates to satisfy needs;

3) Establish benchmarks to compare different languages;

4) Set up a plan for the analysis;

5) Identify in house capabilities to perform this analysis.

This is not all that is covered in this report. Because of the importance of being able to fruitfully compare programming languages and the lack of surveys of this area, several additional topics are presented. In particular, we consider several other language criteria: theoretical, matrix methods and the direct observation of programmers at work. Because of the effort spent in the laying of this extensive groundwork, there was not time to complete the third objective of establishing benchmarks.

SPACE STATION REQUIREMENTS

General Mission Requirements. Some of the key features specified for the Space Station which are pertinent to this discussion are one language, commonality of software, evolutionary growth, flexibilit, for many different tasks such as data handling, scientific computation and real-time control, and, most unusual for an endeavor of this kind, a 30 year lifetime. In more detail, from the Space Station Program Description Document (SSPDD, book 1, p. 3-27, we can pick out those particular features which will be of concern

ORIGINAL PAGE IS OF POOR QUALITY

to programmers. It is mandated that the Space Station:

- Shall assume a phase C/D start by or before FY 1987 to support a flight in the early 1990's;
- Will be shuttle-compatible for delivery, assembly, and disassembly;
- Will be supported by the Shuttle, initially on 90 day cycles;
- Shall have a design goal for indefinite life through on-orbit maintenance, repair, or replacement;
- Shall have modular-evolutionary design that permits growth and accepts new technology;
- Shall consider both the initial development cost and life-cycle costs as a design driver;
- Shall be user-oriented to the maximum extent possible implying flexibility and simplicity of user interface systems and documentation;
- Shall have a design goal of commonality for hardware and software of identical or similar functions in terms of systems, subsystems, and interfaces;
- Shall incorporate on-orbit autonomous operations to minimize crew and ground involvement as a design driver."

MSFC Tasks. Marshall Space Flight Center is responsible for Work Package 1, which includes the common module, the logistics module, the laboratory module, the orbital maneuvering vehicle, and the orbital transfer vehicle. The common module will contain the following systems: data management, electrical power, thermal communications, environmental control and life-support. By outfitting it in various ways the common module can be turned into the logistics, laboratory or other kind of module. The logistics module is the supply depot and will hold fuel and fluids, racks of solid consumables, payloads, and equipment to support transport and transfer. The laboratory module is for technology and manufacturing and is to be outfitted so as to be as simple as possible for the users. Marshall is also responsible for accomodating and servicing the OMV and OTV. For more details, see [SSDPD].

Their Programming Needs. From the contractors' reports([Intl, Table 1 and [Liu], Table 1), we find what is needed in the way of language characteristics to fullfill Work Package 1. Since we will be talking more about these kinds of matrices in a later section, for now we summarize their contents. Needed for Work Package 1, in addition to the usual primitive numerical, character, logical and array data types, and scientific functions built out of these, are machine addressibility, priority control, graphics capability and variable-length records, the last for operating systems. In each these reports, these language characteristics are correlated with the particular systems given above, where they are needed. Interestingly enough, nothing in these lists mentions the need for languages capable of supporting more exotic systems such as robotics or expert systems; we will now address this.

Robotics. Automation and robotics are mandated to play a significant role in the Space Station. No doubt, this will drive the invention of new computing devices, and consequently, new computer languages. For this reason, we briefly review the future of robotics in the Space Station.

The best reference for this purpose is [A&R1]. This report, prepared by the Advanced Technology Advisory Committee of NASA, makes thirteen principal recommendations to NASA, of which four are pertinent to us:

Automation and robotics should be a significant element of the Space Station program;

The initial Space Station should be designed to accommodate evolution and growth in automation and robotics;

The initial Space Station should utilize significant elements of automation and robotics technology;

NASA should provide the measures and assessments to verify the inclusion of automation and robotics in the Space Station.

Then the report on p.11 goes on to propose specific goals in each of the subsytems listed in the a previous chapter; these goals recommend the development of robots, intelligent controllers and expert systems for a variety of specific tasks.

In anticipation of such activity, it would seem prudent to identify appropriate programming languages that will be needed or will enhance such projects. Projects now underway at NASA in automation and robotics are outlined in [NAR].

Artificial Intelligence. Expert systems[McA], such as the Liquid Oxygen Expert System at Kennedy Space Center and others[Gn], appear as the most visible and successful subfield of that amorphous body of assorted knowledge known as artificial intelligence. Whether or not other research now being done in A.I. will evolve over the lifetime of the Space Station into equally useful applications remains to be seen. LISP and Prolog are advocated and used for research and some applications because of the lack of explicit control. This is often a drawback since the resulting large search space causes slow performance. Some, therefore, advocate using these functional logical languages for research and, when the research proves successful, redoing the algorithm in a procedural high-order language. This goes along with the findings of Schwartz and Melliar-Smith[SM, p.1] that Ada, as presently standardized, is unsuitable for artificial intelligence applications, but that, "with relatively modest extensions to Ada within the spirit of the language, it would be possible to translate a substantial proportion of AI algorithms ... into Ada."

CANDIDATE LANGUAGES.

Desideratum. A time scale of thirty years into the future is extraordinary. The opposite, thirty years into the past, by way of comparison, would take us back to 1955, the time of the first Univacs and IBM's. Enormous changes have occured since then. Programming has advanced from assembly language to high-order languages, with major concepts emerging at least every decade: algebraic expressions, control abstraction, data types and modularity (Cf. Ng and Fallenstein in [SSS11]). How can we intelligently plan in the next three decades for breakthroughs yet to come when we probably now have no inkling of what some of them will be? Truly a Herculean task requiring the wisdom of Solomon.

Given that design and development of the space station is to begin in 1987[SSPDD], and that the software design environment should presumably be in place by about then, together with the fact that significant software innovations typically take around ten years to develop from first concept to finished product, then we see that there is not much time for extended experimentation. What is to be done? We should identify those seeds of contemporary software research which are likely to blossom and bear fruit in future languages. Examples might be: equational inference for data types as in OBJ; more generally, relational inference as in PROLOG; and further advances in generic definitions. Whatever languages are chosen must be extensible and leave room for such innovations in the decades to come.

Candidates. We should evaluate and compare these languages: HAL/S, Forth, C, Pascal, Modula-2, Ada, FORTRAN, APL, LISP, and Prolog. The first seven are procedural languages; the last three are functional. FORTRAN and APL are useful in mathematics, engineering and applied mathematics. LISP and Prolog are often seen in artificial intelligence.

We briefly review the principle features of these candidates.

HAL/s. Block-oriented and structured with many modern featues but also some unsafe ones. Good for real-time applications; used only within NASA and its contractors.

FORTH. A small but powerful, extensible

language originally designed for real-time control of telescopes.

C. Designed for systems software; structured but considered lower-level by many. Some programmers find C, as well as Forth, hard to read.

Pascal. Highly popular but with design deficiencies -- less powerful than a Turing machine. Structured but no sparate compilation of program units.

Modula 2. Wirth's worthy successor of Pascal. Capable of multitasking, intended for dedicated computers. Not widely used yet. See [MG] and [Wi].

Ada. For embedded systems; with multitasking; separately compilable and reusable packages; generics.

FORTRAN. Second high order language to be designed. Lacks many features now available on languages designed later, but very widely-used among in engineering circles.

APL. A mathematically oriented interactive language with many powerful primitives. Limited data types make it difficult to use for purposes for which it was not intended.

LISP. The first high order language to be designed. Useful for logically oriented problems. Many versions, the later ones of which are quite powerful.

Prolog. Intended for logic programming, which means working in essentially first order predicate calculus, as opposed to working with functions in LISP. Quite special purpose. See [CM].

ADVANCES IN HARDWARE

As with software, hardware has also seen significant breakthroughs, again about once a decade. It has advanced from vacuum tubes through discrete transistor diode logic to small scale integration and finally to very large scale integration today. What further advances can be seen in a crystal ball? Clearly, further miniaturization of circuits and consequent parallel processing/multiprocessing are in store for us. Possible, but not certain, are wafer size chips and truly three dimensional layouts. What effect these developments will have on language design has been little explored[Bro1], but some conjectures will have to be made. We could chose to ignore these questions, but in as much as room and power are at a premium in the Space Station, it behooves us to incorporate hardware improvements as they become available. As an aside, CAD advances are making it easier and easier for small groups to design their own support chips (e.g., the Berkeley toolkit and silicon compilers for microcomputers). Witness the current efforts of JPL in this regard. Thus, hardware design no longer entirely depends on what chips are commercially available.

CRITERIA

Theoretical Considerations. It is often said that all programming languages are equivalent in the sense that anything which can be computed in one language can be computed in any other. According to this view, any perceived differences are strictly a matter of performance, speed, memory, or ease of programming. When pressed, an advocate of this view will claim that it is obvious that all programming languages are computationally equivalent to a Turing machine. To the contrary, we claim this is not obvious, in fact, it is false for some well known languages, for example, Pascal as original defined by Wirth.

We would also claim that one must clarify what is meant by a programming language being equivalent to a Turing machine. The simplest assertion to try to prove is that all the functions which a Turing machine can compute, namely, the partial recursive funcions, are precisely the ones which the language in question can compute. But this is patently false for Pascal since it does not allow dynamic arrays, and hence cannot even compute the concatenation of an arbitrary array with itself.

But one can pursue this line of inquiry further, and ask what is the common alphabet over which these functions are being computed. This is necessary in order to talk about recursiveness. When input to hardware was a paper tape, the analogy with a Turing machine with its hypothetically infinite tape was easy to see. What is it today: a screen, a disc, a printer? Since interactive computing has become so popular, the screen would seem to be the obvious medium. If any two languages are equivalent, and one can do whatever the other can, then it would follow that it should be possible to write a program in the first language which will make the screen behave in such a way that an unsuspecting programmer would think he was dealing exclusively with the second (this is reminiscent of Turing's test for artificial intelligence). But does anyone really think they could write a Basic program which would present to the unwary programmer sitting at his terminal what would appear to be a bona fide Ada editor, compiler and executor?

So far this is accepting the definition of a Turing machine at face value. But in its definition, there is no provision for formulating concepts having to do with input-output, multitasking, and real-time behaviour. Thus, it really makes no sense to talk of recent high-order languages such as Ada and FORTH as being equivalent to a Turing machine.

Quantitative Benchmarks. In general, a benchmark is a standard or reference point for measurement. In computer science, it refers to a program which has been so written as to compare quantitatively the performance of different computers, compilers or languages. That a benchmark can do all three is a strong point. It is also a weak point in that it is often difficult to isolate what one is really measuring. Let us look at several benchmarks.

An early one is the Gibson mix, which "is a weighted average of instruction times, reflecting usage expected of scientific work."[CW] Because the performance of this mix reflects peculiarities of particular computers, it has been criticized for unduly penalizing computers without pipelining and primitive floating point operations.

The Whetstone benchmark[CW], developed to meet these objections by the Computer Agency of the United Kingdom, "does not solve a real problem; it is a mix of higher order language statements that reflect the frequency of such statements as measured in over 900 actual scientific programs.[Cur]" As such, it is widely used. The result is measured in Whetstone instructions per second(WIPS). Martin-Marietta of Denver, over the last few years, has run this test on a variety of machines, large and small, using an assortment of operating systems and high order languages. But note: the Whetstone does not test multitasking, interrupt handling, or input and output.

Caveats. Designing benchmarks is tricky; we want sensitivity but a particular benchmark may measure something different from what we want. For example, the benchmarks of Kulcin[Kul] are quite sensitive to the degree of optimization of the FORTRAN compiler. If this is what we're after, fine; otherwise we are in trouble. Consult Kulcin's article to see how other factors may skew the metric.

What are the implications of these studies for choosing high order languages? First, there do not appear to be any benchmarks that thoroughly test all the features of the more recently designed languages, such as HAL.S, Modula 2 and Ada. Secondly, different benchmarks will have to be designed to measure the performance of languages dedicated to different tasks, e.g., robotics and artificial intelligence. Here again, in general, there are no ready-made benchmarks. As an exception, however, here at Marshall Yancy[Yan] has used a database benchmark to compare two intelligent database machines.

Also, we mention the paper of Walters[Walt], which discusses what he calls 'efficacy factors' of benchmarks. These are accuracy, sensitivity, effort, and portability. While Walters's comments are addressed mainly to how these affect the evaluation of architectures, much of what he says is transportable to other domains.

See [Levi] for a critical account of other papers discussing benchmarks.

Qualitative Matrices. This is the most popular way to evaluate languages for a particular task. Roughly, one draws up a table with the language candidates at the top, the desired tasks at the side, and the evaluation in the body. This is done in a number of contractors' reports(e.g.,[Int], [Liu])

In practice, the process is more complicated. As an intermediary, one must postulate the programming features which the task needs. A complex project, such as the Space Station, has many systems, i.e., tasks. This leads to two matrices: one with tasks vs. features, and the other with features vs. languages. As an example of this process we reproduce small portions of two matrices, the first from Table 1 on p.5 of [Liu] and the second from Table 2 on p.8 of [Int].

Table 1 - Requirement Levels for Various HOL Features in Space Station Applications

FEATURE/CRITERIA	GNC	ECLSS	ELEC	THRML	C&T	FL	OMS
Float. Pt. Arith.	m	m	m	m	m	m	n
Fixed Pt. Arith.	n	n	n	n	n	u	n
Integer Arith	m	m	m	m	m	m	m
Bit Manipulation	u	m	m	m	m	u	m
Matrix Algebra	m	n	n	n	m	m	n
.						.	
.						.	
.						.	
Data Abstraction	u	u	u	u	u	u	u
Interfacing with other Languages	u	n	n	n	u	u	n

In the preceding table, "n" stands for nil, "u" for useful, and "m" for mandatory.

Table 2 - Language Features Comparison.

FEATURE/LANGUAGE	HAL/S	Ada	JOVIAL	conc.Pascal
Float. Pt. Arith.	D	D	D	D
Fixed Pt. Arith.	D	D	-	-
Integer Arith.	D	D	D	D
Bit Manipulation	D	D	D	-
Matrix Algebra	D	L	L	L
Data Abstraction	--	D		D
.			.	
.			.	
.			.	

In this table, "D" means that the feature is supported by the language, "L" that the feature can be added as a library of subprograms, and "-" that the feature is effectively unavailable. (We have taken the liberty of rearranging this excerpt of Table 2 somewhat so that the same language features appear in both tables.)

As remarked before, these are the easiest kind of comparative studies to do on languages, particularly if one has personnel who are familiar with the tasks and the languages under consideration. Unfortunately, it is very easy for the entries in these matrices, and hence the conclusions of the studies themselves, to be clouded by subjective biases, as has been demonstrated. See [Haug], [Hnor] and [HRW] for other matrix studies.

Direct Evaluation by Programming. The idea here is to set programmers to coding pertinent tasks in the candidate languages. Then observe them at work, time them, and interview them afterwards. On the basis of all this evidence, choose the right language.

One reason for observing programmers at work is that the newer languages are more than just that: they also encompass programming environments. Thus one aspect of the worth of a new language is how well it can be integrated into a productive system.

The method should be the following. Pick tasks typical of Space Station needs. Perhaps consider recoding some of the programs already done on the Shuttle or Skylab. Have programmers of comparable skill and experience put on teams spread among the candidate languages to do a selected sample of tasks typical for the Space Station. Then time the programmers at their work. Obtain compile and run times, check for readability of the resulting code, and ask them their comments, opinions, and recommendations about various aspects of the languages. One can go further and test for maintainability of the langugages by slightly changing the specifications of the tasks which were just programmed to see how easily the coding can be modified.

This method takes some care to get reliable and reproducible comparisons. As remarked already, the programmer teams chosen must be comparable in experience and background. We can't use the same programmers to code the same task in two different languages since, in recoding the task in the second language, the programmers would already know the task and be ahead. A statistical sample of sufficient size would probably mean too many programmers. Nevertheless, we could probably learn much from such an experiment of small size. For example, see [Kle].

PLANS AND COST

Plan for Analysis. In view of the facts that no backup language for Ada has been identified, and that no languages have been singled out for work in robotics, automation, and artificial intelligence, and that only one of the four methods of testing comparing languages has been done in depth for more than a few of the candidate languages, we offer the following advice.

Do theoretical work on the lack of equivalence between various high-order languages. Try to find measures of complexity which actually separate well-known languages according to our intuitive feeling for their relative power.

Develop and establish benchmarks which evaluate advanced programming concepts such as user-defined data types, generics, multi-tasking, modularity and input-output capabilities.

Develop and establish benchmarks for tasks typical of robotics, automation and artificial intelligence.

Perform these benchmark tests on our candidate languages.

Identify language characteristics suitable for robotics and artificial intelligence.

Carefully pick a set of programmers comparable in ability, pick a subset of the candidate languages which have done well in benchmark and matrix tests as well as scoring high on theoretical grounds, and let the programmers try their respective hands at coding typical tasks in the various languages.

This should be an ongoing study. There are several tasks appropriate for future work.

Do the previous tests on languages likely to gain maturity in the near furture, such as Smalltalk and UBO, which are object-oriented languages suitable for work in artificial intelligence.

Do a study anticipating future advances in VLSI impinging on languages and programming.

Cost. We make the following ball-park estimates only for personnel required since most of the hardware and software will soon be in place in the language laboratory at Marshall. We do this for each of the four methods for comparison described earlier.

Theoretical Criteria. One-half a man-year should produce some worthwhile results. Not all ramifications could be followed up in that time, but one should know better then what to pursue.

Quantitative Benchmarks. We distinguish benchmarks with only the usual features, those with advanced features such as user defined data types, and those for artificial intelligence. The cost is given tabularly in man-months.

	usual	advanced	A.I.
design	2	4	[illegible]
coding	4	6	6
consulting and follow up	1	2	1 1/2

Qualitative matrices. Much has already been done with conventional procedural languages. The area in which new work should be done is the languages for robotics and artificial intelligence. Identifying features for systems of the Space Station has already been done[ARTI]. Identifying which special purpose languages have which features should take on the order of a quarter of a man-year.

Direct Evaluation by Programming. A modest effort would be 6 programmers split into teams for six months, for a total of three man years.

In-house Capability. We give brief opinions as to what extent the Systems Software Group at Marshall could perform each of the preceding tasks, both as regards equipment and personnel. With the coming installation of several IBM PC/XT's and a VAX, the Software Evalution Laboratory at Marshall, as well as compilers for a wide variety of languages, the Software Evaluation Laboratory at Marshall will soon be have excellent equipment for performing this research. As to personnel, the situation is less clear, and we break these estimates down again by the four kinds of comparisons.

Theoretical Criteria. There appears to be no one at Marshall with these skills. Perhaps this could be done in academia.

Quantitative Benchmarks. As this would involve mimicking tasks similar to those already done in previous missions, the personnel at Marshall could certainly do this.

Qualitative Matrices. Clearly there are sufficient staff at MSFC in A.I. and robotics to draw up a matrix of desirable language features in these areas. Thus the remaining matrix of features vs. language can be drawn up in house.

Direct Evaluation by Programming. There are more than enough programmers at Marshall to form some matched teams. The only question is whether Marshall can spare so many.

SUMMARY

We have explored how to compare and evaluate a variety of programmer languages for the many-faceted space Station. Out of this study came four tests: theoretical criteria, quantitative benchmarks, qualitative matrices, and direct evaluation by programming. Marshall should select two or three of these methods to pursue in depth. Some should be done at Marshall to develop in-house expertise. Others can be contracted out.

REFERENCES

[ATAC] Advanced Technology Advisory Committee. **Advancing Automation and Robotics Techmology for the Space Station and for the U.S. Economy. Vol. 1 -- Executive Overview.** NASA Technical Memorandum 87566. Submitted to the U.S. Congress April 1. 1985.

[And] Warren Andrews. Silicon solutions begin surfacing for high resolution display problems. **Electronic Engineering Times,** 337 (July 8, 1985), pp. 35-38.

[BEU] Erwin Book; Paul Eggert; Robert Uzgalis. **A Methodology for Evaluating Languages and Their Compilers for Secure Applications.** Command and Control Technical Center TM 171 78, Jan. 31, 1978. AD A05693[illegible].

[Bro] Chappell Brown. Ada system rivals supercomputer speeds. **Electronic Engineering Times,** 337, (July 8, 1985).

[Car] Phillip C. Carney. Selecting on-board satellite computer systems. **Computer** (April 1985), pp. 35 41.

[CM] W. F. Clocksin; C. S. Mellish. **Programming in Prolog.** Springer-Verlag, 1981.

[Cug] John V. Cugini. **Selection and Use of General-purpose Programming Languages -- Overview, vol.1, Programming Examples, vol.2.** National Bureau of Standards Special Publication 500-117.

[CW] H. J. Curnow; B. A. Wichmann. A synthetic benchmark. **The Computer Journal** 19:1 (1976), pp.43-49.

[FG] Alan Feuer, Narain Gehani, eds. **Comparing and Assessing Programming Languages -- Ada, C and Pascal.** Prentice-Hall, Inc. 1984.

[Hor1] Ellis Horowitz. **Fundamentals of Programming Languages,** 2nd Edition. Computer Science Press, 1984.

[Hor2] Ellis Horowitz. **Programming Languages: A Grand Tour.** Computer Science Press, 1983.

[Int] Intermetrics, Inc. **Justification for Selection of a Standard Language for Space Station Applications.** Final Report, 28, June 1985. (Prepared for Rockwell International)

[Kle] Melvin Klerer. **Software Modeling Studies -- Experimental Study of a Two Dimensional Language vs. FORTRAN for First Course Programmers.** Rome Air Development Center, July 1981. RADC-TR-81-183, vol. III.

[Klu] Allan R. Klumpp. Space station flight software: HAL/S or Ada? **Computer** 18:3 (March 1985), pp. 20-28.

[Knos] Alton A. Knosp, Jr. **AIPS Language Trade Study.** Charles Stark Draper Laboratory, Cambridge, Mass. March 1984. CSDL-C-5694.

[Lev] Bruce Leverett. **Performance Evaluation of High-level Language Systems.** Air Force Office of Scientific Research, Nov. 77. AFOSR TR-78-1088.

[Liu] O. Liu. **Assessment of Ada for Space Station Applications Software.** McDonnell Douglas Astronautics Co. July 1, 1985.

[McA] Michael A. McAnulty. Evaluationof Expert Systems. **Research Reports -- NASA/ASEE Summer Faculty Research Fellowship Program,** Marshall Space Flight Center, Jan. 1985. NASA CR-171317.

[MG] Joel McCormack; Richard Gleaves. Modula-2 -- a worthy successor to Pascal. **Byte** (April 1983). pp.385-395.

[MM] Martin-Marietta Corp. **Whetstone Benchmark Test.** (Tests performed 1981-1984).

[NAR] Advanced Technology Advisory Committee. **NASA Automation and Robotics -- Information Exchange Workshop - Proceedings. Vol. 1 - Executive Summary.** Held at Lyndon B. Johnson Space Center, Houston, Texas, May 13-17, 1985. JSC Artificial and Information Sciences Office, June 5,1985.

[Per] Alan Perlis; Frederick Sayward; Mary Shaw (editors). **Software Metrics: An Analysis and Evaluation.** MIT Press, 1981.

[Ful] Jeffrey Fulcini. Constructing benchmarks that measure up. **Computer Design** 22:11 (Oct. 1983). pp.161-168.

[SM] Richard L. Schwartz; P. M. Melliar-Smith. **On the Suitability of Ada For Artificial Intelligence Applications.** SRI Int. July 1980. AD A090790.

[SMB] C. H. Smedema, P. Medema, M. Boasson. **The Programming Languages: Pascal, Modula, CHILL, Ada.** Prentice-Hall, 1983.

[SSDPD] **Space Station Definition and Preliminary Design -- Request for Proposal.** NASA, Sept 15, 1984.

[SSPDD] **Space Station Program Description Document.** NASA, Space Station Task Force, March 1984. NAS 1.15:86652/bl.1-4,6 7.

[SSSI] **Space Station Software Issues.** Report of a workshop held at NASA Langley Research Center, Hampton, Virginia, August 20-21, 1984. NASA Conference Pub. 2361, 1985.

[SSSR] **Space Station Software Recommendations.** Report of an Open Forum with Industry and Academia held at NASA Marshall Space Flight Center, Huntsville, Alabama, April 24-25, 1985.

[TRW] **Language Task 3 Trade Study.** TRW Defense Systems Group, April 1985.

[Walt] R. E. Walters. Benchmark techniques: a constructive approach. **The Computer Journal** 19:1 (Feb.1976), pp.50 55.

[WS] B. A. Wichmann; A. H. J. Sale. **A Pascal Processor Validation Suite.** National Physical Laboratory Report CSU 7, Teddington, Middlesex, U.K. march 1980. ISSN o143/7337.

[Wi] Niklaus Wirth. **Programming in Modula-2,** 2nd edition. Springer-Verlag, 1982.

[Yan] J. L. Yancey. The intelligent database machine. **Research Reports -- The 1984 NASA/ASEE Summer Faculty Research Fellowship Program.** Marshall Space Flight Center, Jan. 1985. NASA CR-171317.