1986

NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

MARSHALL SPACE FLIGHT CENTER
THE UNIVERSITY OF ALABAMA

BENCHMARKS OF PROGRAMMING LANGUAGES FOR SPECIAL PURPOSES IN THE SPACE STATION

Prepared by:	Arthur Knoebel
Academic Rank:	Professor
University and Department:	New Mexico State University
NASA/MSFC:	
Laboratory:	Information and Electronic Systems
Division:	Software and Data Management
Branch:	Systems Software
MSFC Colleagues:	John W. Wolfsberger Robert L. Stevens
Date:	August 1, 1986
Contract No.:	NGT 01-002-099 The University of Alabama

BENCHMARKS OF PROGRAMMING LANGUAGES FOR SPECIAL PURPOSES IN THE SPACE STATION

by

Arthur Knoebel
Professor of Mathematical Sciences

ABSTRACT

Although Ada is likely to be chosen as the principal programming language for the Space Station, certain needs, such as expert systems and robotics, may be better developed in special languages. This report studies the languages, LISP and Prolog, and draws up some bench marks for them. It starts off by reviewing the mathematical foundations for these languages. How this works out in practice is examined briefly. Likely areas of the space station are sought out where automation and robotics might be applicable. Benchmarks are designed which are functional, mathematical, relational, and expert in nature. The coding will depend on the particular versions of the languages which become available for testing.

ACKNOWLEDGEMENTS

Making this summer fellowship successful for all the visiting faculty involved, of necessity, the close cooperation of many people. I myself would like to single out eight of these: my counterparts, John Wolfsberger and Robert Stevens, for their helpfulness; the coordinators of the program, Michael Freeman and Ernestine Cothran, for making it run smoothly; immediate supervisors, Walter Mitchell and David Aichele, for their interest; and higher up, Jack Lucas and Gabriel Wallace, for their efforts beyond the call of duty in arranging worthwhile tours.

TABLE OF CONTENTS

INTRODUCTION

Statement of Problem. The idea of establishing a colony of humans somewhere beyond the earth goes back in fanciful forms to antiquity, takes a more realistic turn earlier in this century, and now assumes many of the features already present in the sketches of von Braun. The first realization of this concept was finally achieved by the Soviets when they launched Salyut in 1971; this space station has been in use, on and off up to the present day, with a recently completed tour of duty for the astronauts of record-breaking length. Sky Lab, the second space station, was put in orbit by the U.S.A. in 1973, used off and on into early 1973, and sometime later disintegrated ignomineously over western Autralia. President Reagan, in his inaugural address of 1984, gave new life to this old notion, announcing that the United States was committed to henceforth designing and building a brand new space staion, and putting it into orbit by the mid 1990's. With this fresh start come fresh decisions to be made about every facet of space voyage, discovery, and habitation.

Playing an important role in these deliberations will be many questions about the software that goes with the Space Station. Software is often called the glue that holds modern technology together. Already intensive debate has started on what kind of software should be developed for the Space Station, and how.

The first question to answer is what language, or languages, should software be written in. Heretofore at NASA, the principal languages have been FORTRAN, HAL/S and assembly, as well as a handful of many others used only slightly. How to choose among all of these was the topic of my report of last summer [Knoe]. Since that time it seems rather certain that Ada, a fairly new and relatively untried language, will be chosen.

Given that, there still remains the second unanswered question as to whether Ada should be the only language allowed, or whether other languages should be allowed for special puposes, such as, expert systems, robotics and perhaps other tasks demanding exotic algorithms. Two typical languages that ought to be considered along this line are LISP and Prolog. This is the principle topic of my investigation.

Four Ways to Choose a Programming Language. There are many ways to choose a language. How we go about this depends on what we want to program, on whether we want to use an existing language or design a new one afresh, and how many resources we have to spend in comparing. Since the coding for the Space Station is to start within a couple of years, we are constrained to choose among existing languages which are fully defined, and now have or will shortly have a software development environment. Last summer I surveyed four methods for comparing languages already at hand; these were theoretical, by matrices, with benchmarks, and through observing programmers. We will now briefly describe these. Toward the end of this report, we will consider how the languages we are going to look at satisfy other criteria proposed for designing languages.

The theoretical method of comparing programming languages is to use the definitions of the languages to find out which is better by trying to calculate relative speeds, memory requirements, and extent of equivalence. Programs themselves are not run. For LISP and Prolog, where combinatorial search strategies often lead to exponential growth in time and memory, this would seem to be an important study to do. Surprisingly, there is little done along these lines, either for these two languages, or in general (e.g., [Haw]).

In the matrix method, one firsts constructs a matrix with the tasks to be performed going down the left side and the languages to be examined going across the top. In the body of the matrix will be estimates of how good the matches are. In practice, usually two or more matrices are used; see [Knoe].

Benchmarks will be the principle method examined in this report. Programs are written to test a variety of features needed to accomplish certain tasks, and then run in a variety of languages on an assortment of machines. Results are compared and the best language wins. This method, despite its straightforward appeal, is not as easy to apply as it might appear, as we shall see in the sequel.

Observing programmers speaks for itself. By setting programmers to work actually writing code for assorted tasks typical of what we want to do, we can gather statistics on the relative ease of programming in one language versus another, how likely are mistakes to be made, and how easy it is to maintain the code. By itself this method can be quite expensive.

This report is organized somewhat differently from that of the previous summer. As benchmarks will be the principal criterion for distinguishing good implementations from bad, we relegate discussion of the other criteria to single paragraphs in other sections where they most naturally come up. Theoretical methods are at the end of the section on languages; matrices at the end of the Space Station section; and observing programmers in the section on other considerations.

Objectives. Our objectives are three in number.

1) Describe the candidate programming languages, LISP and Prolog.

2) Find nontraditional tasks in the Space Station which might fruitfully use these languages.

3) Establish benchmarks.

BACKGROUND IN LOGIC

The two principle languages we shall consider are firmly based on well-defined logics: LISP on the lambda calculus, and Prolog on the first-order predicate calculus. Therefore, we briefly describe these calculi and related systems before describing the languages themselves. This chapter closes with a comparison of implicit and explicit algorithms.

Pure and Applied Logics. Propositional calculus [Mend] is the oldest mathematical logic, invented in the last century by Boole. In this calculus the atomic entities with which we work are sentences, which will not be decomposed nor analyzed any further. The only attribute of them which we use is their truth value. Thus may we abbreviate the sentence, 'Mary likes John', by the symbol P, and the sentence 'John runs away.' by Q. We assume both P and Q have truth values, but they may not be known to us. These sentences may be combined with what are called logical connectives: for example, P & Q, i.e., 'Mary likes John, and John runs away.' An important connective, material implication, yields a sentence like P => Q, i.e., 'If Mary likes John, then John runs away.' This connective, =>, is the backbone of Prolog syntax, to be explained

later. It is called material implication since in everyday English, 'If-then' statements often imply semantical cause and effect, whereas our logical implication is defined strictly in terms of truth values by a truth table:

	T	F
T	T	F
F	T	T

Compound statements are possible, for example, (P & Q) => S, where S might be, say, 'Mary likes Sam.' The algebraic study of selected connectives is called Boolean algebra, and it plays a crucial role in the design of digital circuits.

Predicate calculus [Mend] is based on the propositional calculus, and is obtained from it by introducing individuals and allowing the logic to recognize some limited internal structure within the propositions, namely, they may be relations, i.e., predicates over individuals. For example, the relation among numbers of 'less than' is a binary predicate. An instance of this would be $3 < 5$; but we also have more generally $x < y$. Predicates may have any number of arguments. Quantifiers are also a part of the predicate calculus: 'for all x' and 'there is an x'. Introducing M(x) for 'Mary likes x', and R(x) for 'x runs away', and also N(x) for 'x is a man', we may paraphrase the formula,

For all x, M(x) => R(x),

as 'Any man that Mary likes always runs away'.

Typically, for each of the two calculi introduced so far, a finite number of axioms and rules of inference are formulated which capture exactly the nature of the these logical structures, and nothing else. Such logics are called pure. If we add axioms asserting something about the world beyond its purely logical aspects, then we have what is called an applied calculus. For example, if the previous formula,

For all x, M(x) => R(x),

is added as an axiom, expressing the fact that Mary scares men away, then we would have an applied calculus. An important applied calculus which everyone has met is Euclidean geometry. Another is axiomatic set theory, which is important since it serves as a foundation for mathematics.

Both of these pure calculi have the desirable metamathematical property of completeness: any formula which is true is always provable from the axioms. This implies in turn that these systems are decidable, i.e., there is an algorithm, programmable on a digital computer,

if you like, which takes a single formula as input and gives as output whether this formula is true or false, or equivalently, provable or not. Unfortunately, applied systems may not be complete and decidable. In fact, Kurt Goedel showed on the contrary that any system rich enough to define the integers will, of necessity, contain formulas which are undecidable. This will have important consequences for our discussion of the limitations of artificial intelligence, to come in a later chapter.

Functional Calculi. We first look at the lambda calculus [Curry], which is built out of two simple but powerful constructs. The first is functional evaluation, f(x). Since everything in this calculus is considered to be a function, regardless of its purpose, any two objects may be combined in this way, for example, also x(f). The other construction is used for defining new functions, lambda x e(x), where e is some expression in the calculus constructed out of objects already available. This expression yields a new function, which, when evaluated, say, f(a), gives the expression e changed by replacing x by a. With the lambda construction, all sorts of new functions may be created. In fact, this extraordinarily small calculus is quite capable of defining the integers and much more. This all assumes that one has formulated the natural axioms and rules of deduction, which we do not have the space to do here.

There is another functional foundation for mathematics, the category theory, which is quite different from the lambda calculus. Its fundamental primitive is composition of functions, rather than evaluation. Again everything is a function, but there are constraints on when functions may be composed. Although many mathematicians feel category theory is the most natural foundation for doing mathematics, surprisingly no programming language has been built around it.

Explicit versus Implicit Algorithms. Traditional computer programs are procedural in the sense that every step of the algorithm must be spelled out. In fact, originally one had to code in assembly language. Now, languages such as LISP and Prolog allow a programmer to specify what is desired without having to give step-by-step procedures. We can spot a trend here towards less and less detailed recipes for accomplishing one's goals. Curiously, just the opposite is happening in mathematics. Most contemporary mathematicians are quite satisfied with an axiomatic presentation of their subject, assuming that whatever objects are needed do indeed exist, without a care

in the world about how they might actually be constructed. But there is a small movement afoot, called constructive mathematics, which has as its aim the discovery and development of algorithms for all classical definitions and theorems which assert the existence of something. It is ironic that while some mathematicians are moving from the abstract to the concrete, computer scientists appear to be moving in the opposite direction.

CANDIDATE LANGUAGES

In this section, we talk about LISP and Prolog. We could have discussed other languages for similar purposes, e.g., Smalltalk, Obj , Nial, but did not.

LISP. Pure LISP is based directly on the lambda calculus. In theory, as a programming language, it is equivalent in computing power to a Turing machine. However, with only a few primitive constructs, it is of necessity tedious to code. For that reason a variety of extensions are made, which lead to the applied LISP's (see [Gab]).

All LISP's commercially available today have a central core of pure LISP together with quite a few extensions, the number and kind depending on the particular version. Most have the integers defined, as well as conditional clauses and loops. There is usually a programming environment containing various debugging and program development tools.

How a LISP dialect is implemented is crucial. The theory tells us that are two principl ways in which lambda expressions may be evaluated. One always converges but can be quite slow. The other is usually faster when it converges, but there is no guarantee that it will converge. All implementations use one or the other; some both, allowing the programmer to choose.

The two most important drawbacks of LISP are its slow execution speed and extensive demands on memory. Two interrelated hardware solutions have been proposed. The first is to design digital computers for processing LISP programs out of existing off-the-shelf chips. The second is to design and fabricate new chips themselves [Line].

Prolog. Prolog is based on the first-order predicate calculus, but not directly. The only acceptable formula is of the form

R1 :- S1, S2, ...

where R1, R2, , ..., S2, ..., S2 are relations in variables and constants. In our syntax of the predicate calculus this Horn clause would read

(S1 & S2 & ...) => R1

That this drastically reduced syntax can express much of the predicate calculus depends on three tricks:

i) there is implicit universal quantification, that is, each Horn clause is universally quantified over all variables occuring in it;

ii) existential quantification is achieved by the judicious use of constants;

iii) sets of Horn clauses are allowed, which has the effect of 'and'ing them together.

A nice feature of Prolog is that it can prove statements and answer questions. However, Prolog is not completely equivalent in expressive power to the first-order predicate calculus (cf. [HC]).

The execution of a Prolog program seeks matches of symbolic patterns, and unifies them. Again we have a choice of algorithms for evaluating these sets of Horn clauses, some slow but guaranteeing convergence, and others fast when they terminate. The problem is particularly acute in Prolog since some kind of backtracking while traversing a tree is necessary in order to explore all possibilities. Needless to say the demands on memory are enormous. To help alleviate this problem, and to give programmers some control over the course of execution, the cut, a feature unique to Prolog, is to be found in every version of Prolog.

Their are many dialects of Prolog, and [Camp] describes some of their flavors.

One feature no version of Prolog has today is the implementation of cyclical instantiation. This is a somewhat technical substitution sometimes needed in the theory to insure termination. It is not implemented because run times would be much too long. Opinion has it that this situation would never occur in real programs. However, to me, it appears to be a condition that would appear quite frequently in automatic theorem proving. In view of the closeness, in theory, of the spirit of expert systems with theorem proven, one has to wonder how valid current implementations of Prolog are for work in artificial intelligence.

Prolog is the darling of the fifth generation project in Japan [FM]. Rumor has it that Prolog was invented by the Europeans and given to the Americans as a joke, but the Japanese have yet to get the point. Certainly, a variety of substantial problems with Prolog will have to be solved before it can be widely used in reasoning systems. Chief among these are its slow speed, its massive memory requirements, and its extraordinary sensitivity to the order in which clauses are presented.

Theoretical Comparison. All versions of both LISP and Prolog should be equivalent in computing power to a Turing machine, i.e., both can compute all partial recursive functions. It would be nice also to have complexity comparisons for both speed and space. See [GB] for a more down-to-earth comparison.

AUTOMATION AND ROBOTICS

Artificial Intelligence. This phrase refers to efforts to program digital computers to do certain human activities which require intelligence in the common sense of that word. As such it is extraordinarily nebulous; henceforth we avoid the term 'artificial intelligence', and instead talk about the specific human activities we are trying to mimic by a computer. Examples might be playing games such as checkers and chess, poker and bridge; professional practice such as medical diagnosis and treatment, engineering design, and preparation of law briefs; translating natural languages and nontechnical interfaces between humans and computers; creative work such as composing music and proving theorems.

It would be good to compare this list of undisputed intellectual activities with what has already been programmed. Very large databases now exist which can be queried in rather sophisticated ways. There are numerical algorithms for solving partial differential equations and large systems of linear equations and inequalities, which perform large numbers of arithmetic operations and make decisions about many different kinds of branches in the computation. Computer-aided design has become extremely versatile, and indispensable in such fields as very large-scale integrated circuits on silicon chips. Packages, such as Macsyma and MPS, can perform an amazing range of symbolic manipulations, as, for example, indefinite integration well beyond what even most mathematicians can perform

straightaway. Are these examples of true intellectual activity? Well, at one time, they would have been considered so. That many no longer consider them activities requiring the donning of one's thinking cap would appear to be a good illustration the of the fact that once something is programmed, it is no longer considered intellectually challenging, and hence not a true instance of artificial intelligence . That is, once the glamour has been removed by the hard binary code of success, our attention shifts to tasks of the mind yet to be conquered.

Each of the successful programs just mentioned in the preceding paragraph has, as a foundation, a mathematical theory which guarantees that the algorithm will work as expected. On the contrary, for the tasks enumerated in the first paragraph of this section, there are no algorithms yet known which will guarantee convergence of the coding in a timely fashion. We say 'timely' since, for example, there are complete axiomatic mathematical systems for which there are exhaustive algorithms, too time consuming to use in practice, but nevertheless they converge. Because there is no theory underpinning the current efforts in artificial intelligence, we should expect, as is typical in any engineering enterprise without a firm foundation, slow progress and few successes.

Caveats. Continually rising expectations by workers in the field of artificial intelligence have led to inflated claims. Listen to this quote from the book by Feigenbaum and McCorduck: 'In the kind of intelligent system envisioned by the designers of the Fifth Generation, speed and processing power will be increased dramatically; but more important, the machines will have reasoning power: they will automatically engineer vast amounts of knowledge to serve whatever purpose humans propose, from medical diagnosis to product design, from management decisions to education' [FM, p.56]. By way of contrast, there have been very few clear successes. We have the gloomy prognosis of the Dreyfus brothers: 'After 25 years Artificial Intelligence has failed to live up to its promise and there is no evidence that it ever will' [DDa, p. 42]. Their sobering critique [DDb] documents this. See also [Bolt] and [Mart].

Among the better programs are Puff, R1, and LOX. Puff, a diagnostic program for lung diseases, is correct about 75% of the time; as such its principal use is to shorten the time spent on preparing reports, when an independent diagnostician verifies that what the system has printed out is correct. R1 matches user requirements for a new computing system with devices available from Digital

Equipment Corporation; it is has proven successful, mainly due to its power to sort through large numbers of combinatorial possibilities. LOX, a creation of Kennedy Space Flight Center [NAR, p. 27] creates tailor-made drawings from a huge database about liquid oxygen pumping systems. It is also designed to diagnose faults but it has yet to be tried in that capacity. Larry Wos, at Argonne Laboratories, has written a mathematical theorem prover [AMS], which has interactively solved three small open problems in abstract algebra. From all of this, it would appear that in the near future the most likely successes will be in interactive expert systems and mathematical theorem proving. These two fields are closer together than one might expect at first glance, since they both use principles of resolution to solve systems of sentences and symbolic equations.

Expert Systems. These are systems which reduce the knowledge and experience of an expert to a collection of rules, often in the form of Horn clauses. Inferences can be made and conclusions deduced using these rules of 'thumb' in the framework of some deduction system such as LISP or Prolog.

In analogy with Goedel's theorem in mathematical logic, it is quite possible that the clauses defining an expert system may not be complete; in which case it will be impossible to answer all queries for lack of information. This is a serious drawback. Since, typically in drawing up an expert system, one has not done a complete logical analysis as one would normally do in more conventional procedural programming, there is no good way to detect incompleteness.

Robotics. It is useful in the discussion to distinguish two main areas of robotics. First there is the present applications, such as welding parts of automobiles together, which involve highly repetitive tasks in a fixed and controlled location, based on a well understood technology. The other kind is postulated to be able to perform in the future a wide variety of tasks, such as building a space station and repairing faulty parts in a highly variable environment.

The first kind of robotics is, of course, now common in a number of industries in various parts of the world. The possibility of making the second kind work is based on a nonexistent theory; it is certainly not an extrapolation of the first kind.

The conclusion we should draw about robotics is that any advances in the near future will be modest extensions of what is presently done. In more advanced systems, close human interaction and supervision will be the norm. See [Whit] for a detailed discussion of this issue.

We summarize this whole section by noting that all algorithms can be classified into four categories, graded according to the increasing degree of 'intelligence' expected:

traditional procedural;
sophisticated procedural;
minimal expert systems;
unreserved AI.

In the next section we will take a closer look at some of the tasks in the Space Station which will fit into the middle two categories.

SPACE STATION

Overview. A permanent presence in space is the next logical step beyond the present short missions, each typically a week long. This is the rationale for the Space Station, which is scheduled to be put into orbit in 1992 and to last for twenty to thirty years. The completed space station will consist of a number of modules for habitation, experiments, and storage. In addition, there will be an orbital maneuvering vehicle and orbital transfer vehicle for moving satellites and astronauts between the space station and a variety of orbits. It is anticipated that that Space Station will be used for everything from zero gravity experiments through servicing of satellites to the launching of probes to the outer planets.

Computer programs will control and regulate all of this and provide timely information. Because of its large size and the long time it is expected to be in orbit, the Space Station will need to be semi-autonomous to be economically run. A special mandate was given by Congress to incorporate automation and robotics as much as possible; ten per cent of the budget for the Space Station is to be devoted to such activities . Thus, it seems reasonable to consider special languages which would be particularly adapted to these tasks.

Timing is crucial. Without going into the details of NASA's schedule for the Space Station, it suffices to note

that detailed design is to begin early in 1987. Therefore, whatever languages will eventually be used must now be available, together with some base of experience in using them.

We now discuss briefly some of the kinds of tasks in the Space Station needing advanced programming techniques. This is difficult since, as seen in the previous section, the techniques of artificial intelligence have seen very limited success. We describe only some of those which will likely fall into one of the two middle categories described in the last section; sophisticated, but traditional procedural programming; and minimal expert systems.

Tasks for Sophisticated Procedures. We see this as an extension of present work along the following lines, where further research and development into sophisticated procedural programming will result in less people needed for day-to-day operations:

Robotics, as for example, parts handling and assembly;
Fault monitoring and diagnosis;
Scheduling of power and load shedding;
Environmental control.

Tasks for Expert Systems. In view of the facts that expert systems are best limited now to manipulating large databases of information in conjunction with an expert, and there can be at most eight experts on board, it would appear that most applications of expert systems will be ground based, for example, LOX, power systems, etc. Those requiring little expertise but lots of combinatorial searching, such as logistics, might well be in the space station itself.

This is only a sampling. For many additional tasks which will fall into these categories, look up principally [AART2], and also [ARP], [Firs] and [NAR].

Matrices. We mention this method of evaluating languages for specific tasks at this point, since it depends on a detailed knowledge of the Space Station. Basically, in previous evaluations (see [Knoe], pp. 13-14), this method constructs two matrices: one to plot tasks versus what is needed in the way of language constructs; and the second to plot language constructs versus languages under consideration. In the present situation, we would recommend three matrices altogether, obtained by splitting

the first one into two. The first matrix would now only plot the matching of tasks versus successful paradigms in automation and robotics; the intermediate matrix would plot these paradigms versus potential language constructs.

BENCHMARKS

We now proceed to the central section of this report: the creation of benchmarks for languages likely to be used in automation and robotics. We split these up into four categories: functional, mathematical, relational and expert. In each category are three benchmarks, for a total of twelve. After motivating each algorithm, we describe it in either English or mathematical terms. Some of these algorithms have been coded into LISP and run, none into Prolog. We leave the completion of the coding to a future project.

Throughout this section we borrow freely from the book of Richard Gabriel [Gab]. Some of the benchmarks of this section are taken directly from his work. He has timings for an impressive array of computing machines.

Designing benchmarks is tricky; we want sensitivity, but a particular benchmark may measure something different from what we want. Gabriel has much to say on this; here is part of his summary (p. 275): 'To claim that a single benchmark is a uniform indicator of worth for a particular machine in relation to others is not a proper use of benchmarking. ... Computer architectures have become so complex that it is often difficult to analyze program behavior in the absence of a set of benchmarks to guide that analysis'. See also my previous report [Knoe, p. XXVI-12] for other comments.

Another difficulty that arises in this study is that the various implementations of LISP are different enough from each other so as to require substantially different coding of the same algorithm. Based on a different theory, Prolog will magnify this problem even more.

Functional. **Ackermann.** The first algorithm comes directly out of arithmetic: we assume all variables are positive integers. It is named after a two-argument function invented by Ackermann, which is doubly recursive but not singly (= primitive) recursive. We give instead a three-argument function which is easier to understand, but

has similar properties..

To motivate it, first observe that multiplication may be defined recursively in terms of addition:

$$m * 1 = m,$$
$$m * (n + 1) = (m * n) + m.$$

Similarly, taking exponents may be defined recursively in terms of multiplication:

$$m ** 1 = m,$$
$$m ** (n + 1) = (m ** n) * m.$$

These may be combined to yield a function A(k, m, n) of three arguments:

$$A(1, m, 1) = m + 1,$$
$$A(k, m, 1) = m \qquad (k > 1),$$
$$A(k+1, m, n+1) = A(k, m, A(k+1, m, n)).$$

Note that this is a doubly recursive definition, and needs only the successor function, m + 1, as a seed function. For small fixed values of k, we get the usual arithmetic operations:

$$A(1, m, n) = m + n,$$
$$A(2, m, n) = m * n,$$
$$A(3, m, n) = m ** n.$$

Thus A(k, -, -) itself may be considered to be a sequence of progressively more involved binary operations. For example, when k = 4, we get iterated exponentiation. The number of iterations is determined by n, for example,

$$A(4, 2, 5) = 2 ** (2 ** (2 ** (2 ** 2)))).$$

The obvious way to evaluate A(k, m, n) for specific arguments is to start with both the first and third arguments set to 1 and iterate to raise to the required values. Unfortunately, as there is no way to predict a priori how large the intermediate values will be, it is necessary to work backwards and first map out the path of computation. This can be quite long and involved, and so this algorithm is good for testing the recursive capabilities of a language.

Polynomial. This program computes specific powers of three trinomials:

x + y + z + 1 ;
100,000x + 100,000y + 100,000z + 100,000 ;
1.0x + 1.0y + 1.0z + 1.0 .

The powers computed are the second, the fifth, the tenth, and the fifteenth. The code, which is five pages long, comes from Richard Fateman, and we are reporting from its presentation in Peter Gabriel's book [Gab, p. 240]. The meters show that the largest number of operations used are those having to do with putting together and breaking up lists. This may not seem surprising since we are using LISP, a list processing language. However, quite roughly less than half this number of arithmetic operations is needed. Clearly, representing polynomials and manipulating them is consuming lots of computing. The representation used in this code is rather awkward, and I can't help but wonder if this may not be slowing down the program.

Raw times for each polynomial and each power are tabulated for a bewildering variety of machines and dialects of LISP. This data, which runs for over 20 pages, is a smorgasbord of food for thought. We content ourselves with one observation on it. The number of LISP operations executed for the fifteenth power is about seven times that for the the tenth powers; this is reflected in some of these statistics, but not all. For example, for a Cray running Portable Standard LISP the CPU times for these two powers are .95 s versus .14 s, a factor of 7. But why, for the VAX 750 also running Portable Standard LISP, is it 82 s versus 4.5 s, a factor of about 14, which is almost double?

Series. I propose the evaluation of the power series for the exponential function as a simple test of the ability of a compiler to sequence assertions so as to truncate the series at the appropriate spot. Here exp(n,x) will be a function of two argument: the first a nonnegative integer, the second a real number. The assertions are

```
exp(0, x)   =  1
exp(n, x)   =  exp(n - 1, x) + t(n)
     t(n)   =  (x ** n) / n!
!t(n + 1)!  >  !t(n + 2)!
     error  =  !f(n + 1)!
     error  <  .01
         x  =  -10.
```

The first three lines define recursively the power series for the exponential truncated at the nth term; the next three lines, the allowable error. We invoke here the theorem which says that if any alternating series has terms

decreasing in absolute value, then it converges, and the error is less in absolute value than the first term to be truncated. Of course, the terms in our series for our particular value of x grow initially; hence we need the fourth condition above, which checks that the terms have begun to shrink.

I have given these assertions in a form directly translatable into Prolog. I don't think LISP can handle this, without substantial modification.

Mathematical. Under this heading, we present three algorithms which make a variety of demands on the languages being evaluated. The first one, a fast Fourier transform tests the floating point capabilities. The second, which calculates derivatives, demands symbolic manipulation. And the third, a theorem prover, asks for pattern matching and lots of inferences.

Fourier. Written by Harry Barrow, this program out of [Gab, p. 193] is a 1024-point, complex, fast Fourier transform. Almost two thirds of the operations consist of what Gabriel, in his meter for this benchmark, calls 'hacking arrays of floating-point numbers'. The mathematical virtue of decomposing a complicated problem into simpler ones really shines in the FFT, as presented in this meter: out of over three million LISP operations, only 200 are for computing trigonometric functions. Of course, there are lots of multiplications. One might expect general purpose machines to perform better on this than machines devoted to LISP, but curiously this not the case. This is seen in this brief excerpt of the CPU times for just the Symbolics and IBM 3081, from Gabriel's tabulation of raw times.

	FFT	Triangle
Sym. 3600	4.75	152
IBM 3081	7.30	25

Derivative. The first part of this program, as reported in [Gab, p. 170], is a general program which takes symbolic derivatives of polynomials. The second part is specialized to taking the derivative of a specific quadratic 5,000 times, in order to get reliable run times. In it the most frequently used operation of LISP is the CON's operation for concatenating lists. Unexpectedly, from Gabriel's timings, we see that the IBM 3081 edges out the

Cray on this benchmark. Also, the spread of timings over all the machines examined appears to be less than that for many of the other benchmarks. Without looking at the execution in more detail, it is hard to know what to make of these anomalies.

Boyer. Ostensibly this is a theorem prover. Reluctantly, I include this benchmark from Gabriel's book, and the reluctance is for three reasons. First, on page 129 in the analysis section, in the set of five statements to be proven, there is clearly an undesirable mixing of types: the function f must operate on both numbers and lists of mixed objects. Although this may technically be allowable in the particular LISP under consideration, intuitively I feel that the implication to be proven should be thrown out on grounds that it is meaningless. Of course, if one only looks at the compound implication itself, then it is a tautology. And it might be accepted by some theorem provers on that ground alone. But shouldn't a really good prover reject sets of statements which are inconsistently typed as data?

Second, these five statements I'm quibbling about are a botched up and substantially incorrect translation into everyday technical prose of that portion of the code which they come from on the previous page of Gabriel's book. (Even if translated correctly this would not invalidate the first objection.)

Third, the LISP program itself runs 13 pages with nary a comment; the definition of the setup function which does the rewriting of expressions is 9 pages just by itself. Moreover, clauses for the same operator are scattered about in no apparent order, so it is hard to verify the content of the operator's definition.

Why then, should one look at this particular benchmark at all? Well, some benchmark in theorem proving must be considered, either this or some other program. Suitability for pattern matching and unification is crucial for any language in AI. Perhaps one should look through the symposium [AMS] of the American Mathematical Society for some better examples of theorem proving.

Relational. These three benchmarks are grouped together here since they can, and should be, coded in some kind of relational data type. The ability to handle relational structures efficiently and quickly is the hallmark of an intelligent programming language. But, as

we shall see in the first example below, this is not always the case.

Triangle. This is a simple puzzle played on a triangular board with 15 holes and 14 pegs, and sometimes found on tables of restaurants, for patrons to while away the time while waiting for their food. A peg may be jumped over another one, as in checkers, providing the landing spot is empty; the peg jumped over is removed. The object is to find a sequence of jumps so that one ends up with only one peg. If, to begin with, one of the three inner holes is empty, then Gabriel, [Gab, p. 219], claims there are 775 solutions; but this number is suspect, since by symmetry there should be an even number of solutions.

Once one has forced the two-dimensional triangle into one-dimensional arrays, the actual writing of the code in Common LISP goes quite naturally. The CPU times run all the way from 14.44 s on the Cray, which is usually the winner in all of Gabriel's bench marks, to 2,866 s on a VAX 730. On all of the VAX's for which code was run, Portable Standard LISP is almost twice as fast as Common LISP. Code for InterLISP is also given, but curiously it is almost twice as long as the code for Common LISP.

We comment further on Gabriel's 'data structure' for the triangle with five holes on a side. This is represented as a vector of length fifteen; to code possible jumps requires three more vectors of longer length, whose entries require some hand computation to determine. A much more natural data type would use barycentric coordinates on the triangle. Possible moves could then easily be calculated from these coordinates. Also, it would now be much easier to scale up from a triangle with 5 holes on a side to one with, say, 10. The only difficulty with this is that LISP has no provisions for the specification of abstract data types such as this. In the final section, I will have more to say about this.

Traverse. This program appears in [Gab] on p. 153, and is called 'traverse' there. It utilizes the record, the least structured data type of LISP, to build a tree at random, and then traverse it. The tree has 100 nodes, and each node is a record with 10 slots for information about parents and sons, and other facts.

Again, in the run times there is a surprise. This time, the VAX 730, running Common LISP, is not behind everyone else; it has edged out the Zerox Dolphin for the

initialization part of this algorithm. However the Dolphin regains its honor during the traverse itself.

One last note. It would be interesting to know how well this algorithm might perform when written in IQLISP, since this language does not contain the record as a data type. Unfortunately, Gabriel does not include IQLISP in his candidate languages. (IQLISP is a dialect intended for personal computers.)

Database. This example is motivated by the likely use of databases in the Space Station, and the setting up of sophisticated query languages capable of making subtle inferences. First we must set up relations; these might be arrays. But, since keys are an integral part of a database, relations would be better represented by records of records.

Next we want to add the four fundamental operations of selection, projection, union and join. Then we should ask both direct queries, and indirect queries phrased as sets of implicit statements. See [Gray] for possibly some sample databases and queries.

Expert. Many workers in automation believe that the most likely area of this subject to be useful to the Space Station will be expert systems. How useful will LISP and Prolog be for these? We propose three small programs to test this. They are Browse, LOX and Power.

Browse. This program is well summarized by Gabriel, p. 139: "This program is intended to perform many of the operations that a simple expert system might perform. There is a simple pattern matcher that uses the form of a symbol to determine its role within a pattern, and the data base of 'units' or 'frames' is implemented as property lists. In some ways this benchmark duplicates some of the operations in Boyer, but it is designed to perform a mixture of operations in proportions more nearly like those in real expert systems."

We note that although the actual program only takes three pages of LISP code, this is tight code, and it takes almost three pages to describe what is does in English. Thus we refer the reader to [Gab] for details.

The meter counting the number of operations invoked and the raw times read differently from the scores of most of

the other programs of Gabriel. For example, testing for equality in various guises is the second most frequently used operation (list manipulation is the first, naturally). Because of this, perhaps, the IBM 3081 outshone the Cray by about 30%.

LOX. The apparatus for feeding liquid oxygen to the combustion chambers from its storage tank is an extremely complex system of piping, pumps, and premixers. As pure oxygen, even in liquid form, is extremely corrosive and explosive, it is essential to have a method of quickly diagnosing faults detected by sensors. To this end the LOX expert system was designed at Kennnedy Space Center (see [NAR, p. 27].

What is proposed here in this report is a miniature of this system to test on different dialects of LISP and Prolog.

Power. The Space Station will require an extensive system for power generation and distribution. Several voltages will be needed, as well as both direct and alternating current. We can expect to see fluctuations in both generation and consumption, and must also plan for various kinds of failures, including outage. The need for intelligent and quick load shedding is obvious.

David Weeks, in this laboratory at Marshall S.F.C., is directing the construction of a model to simulate various aspects of the power plant for the Space Station. Again, it is proposed to code a miniature version of this model in order to compare various versions of the languages which might be used to write an expert system.

Closing Comments. We end this section with a few comments about Richard Gabriel's book, which we have leaned on so heavily. As the first book to publish benchmarks and their run times for Lisp dialects, it is clearly a landmark. Nothing has yet been done for Prolog; nor are there yet any quantitative comparisons of the two languages. Certainly, more such work should be done, and Gabriel's book will serve well as a base for any future efforts.

With this said, it must seem to the reader to be carping to criticize certain aspects of his effort. But this book is difficult to read, even allowing for the great amount of detail presented. Continually changing acronyms and nonstandard abbreviations made it hard for one to be

sure of what was meant, and this slowed a person down. However, this aspect alone could have been surmounted. But there were also real inconsistencies; and so it was hard to trace down their source, and I was left puzzled on quite a few of the details. A good, tough editor would have made the book better.

As I understand it, the main selling point of languages such as LISP and Prolog is that they're are supposed to be easier to program, and the finished code is also supposed to be easier to debug, maintain and modify. Therefore, as remarked earlier, it is always surprising whenever one finds code written in one of these languages which is baroque and both hard to decipher and modify. A good example, as remarked earlier, was the lack of appropriate data types in the program Triangle. In general, one ought to be able to code in a fairly direct fashion and close to the spirit and structure of the original problem domain. And many high-order procedural languages are coming closer and closer to this ideal. So it behooves programmers in nonprocedural languages to do likewise.

OTHER CONSIDERATIONS

This chapter is devoted to gathering together a number of loose ends. We talk about benchmarks in robotics and how to observe programmers, then draw up desirable characteristics which programming languages should possess, and finally discuss how Ada may fit into the picture.

Examples from Robotics. We should have included in the proposed benchmarks some from the field of robotics, since extensive use of these creatures will be made in the Space Station. For having failed to do this, we plead ignorance of where to begin, lack of time to pursue it, and lack of space to report it. This should certainly be done in the future.

Observing Programmers. This is one of the methods for comparing programming languages recommended in my report of last year. This is expensive if done alone. But it should be cheap if done in conjunction with the development of the code for the algorithms proposed in the previous section.

Other Characteristics. Besides speed, ease of programming, and suitability for the tasks at hand, there are many other desirable characteristics of a programming language which have been proposed. We turn to Barbara Lipson's list, as reported by [Hor1, pp. 35-40]. The characteristic is stated, and then sometimes it is followed by a comment or two applicable to the languages we are studying.

1. A well-defined syntactic and semantic description. The pure versions of LISP and Prolog both have this since they have simple, theoretical bases. In their applied forms, with many mixed features, particularly LISP, this is not so clear.
2. Reliability.
3. Fast translation.
4. Efficient object code.
5. Independence of features
6. Machine independence. This seems as far away as ever, particular for nonprocedural languages, where machine-dependent dialects abound.
7. Provability. This is of dubious value, since it requires automatic theorem proving, which is one the things we are trying to accomplish with these languages, but has not been accomplished yet.
8. Generality. This means that there should be just a few basic concepts. This is true only for the pure versions of LISP and Prolog.
9. Consistency with commonly used notations. One can only ask why nonstandard notations were chosen for some symbols commonly accepted and long used in mathematics and logic.
10. Subsets. It's true that subsets can be used in these languages, but only at the risk of writing inefficient programs.
11. Uniformity. Similar things should have similar meanings.
12. Extensibility. Certainly true for common LISP with its ability to use incorporate routines from other languages.

Ada. So far we have said little about how Ada will fit in with this special purposes languages. Several questions arise: how much general, indirect recursiveness does Ada support; how good is Ada for tasks in automation and robotics; and how fast would it be for such tasks. Certainly the standardization of Ada, its multitasking capabilities and the software development environments being created are all good reasons for seriously considering it.

There is increasing evidence [SM] that the best route to programming automation in the Space Station is to develop algorithms and programs in LISP or Prolog, and then to use Ada as a production language. We mean here that the peculiarities of the candidates languages should be taken into account. LISP and Prolog are stimulating for research and excellent for prototypes. Ada may well be better for the final code and implementation.

CONCLUSIONS and RECOMMENDATIONS

1. There will be limited use for the techniques of artificial intelligence in the Space Station. Identify and isolate those subsystems amenable to this method of programming.

2. In anticipation of some use, the benchmarks outlined in this report should be coded, and run in a limited number of dialects of LISP and Prolog. Additional routines should be created to test capabilities in robotics. In the process, programmers should be observed to see which dialects are easiest to use in practice.

3. For actual tasks in the Space Station where inference techniques are warranted, develop prototypes in LISP or Prolog, but after development, transfer the algorithms gracefully to Ada, where speed, reliability and maintenance are important.

4. Identify those enhancements of Ada which will make this transition easier.

5. Even when writing programs in Ada from the outset, isolate data from procedures as much as possible. Be broad-minded about what is considered data. There is much in Ada to encourage this; but special effort is required to reap the benefits of reusable software. Also, make full use in Ada of the ability to create new data types so as to make the code transparent.

6. Design and study new nonprocedural languages based on logical and mathematical principles significantly different from those of the predicate and lambda calculi.

7. Compare languages theoretical in an effort to understand the speed and space limitations inherent in nonprocedural languages. The languages to be compared would include not only LISP and Prolog, but also any new

languages thought of in connection with item 6 above. Hopefully this will lead to new and better interpreters, insuring convergence in a timely manner.

A Final Quote

We end with a quote, which the reader is free to interpret either: as a slur on nonprocedural algorithms which take too long to end; as a comment by a faculty fellow on the anguish of getting a summer project finished in ten weeks; or, more nobly, as a sigh of empathy for NASA's travails in meeting tight launch schedules.

'He that runs against Time has an antagonist not subject to casualties.'

Samuel Johnson

REFERENCES

[AART2] Advanced Technology Advisory Committee. **Advancing Automation and Robotics Technology for the Space Station and for the U.S. Economy.** Progress Report 2 -- October 1985 through March 1986. NASA Technical Memorandum 88785. Submitted to U.S. Congress, April 1, 1986.

[AMS] American Mathematical Society. **Symposium on Mathematical Theorem Proving.** 1984.

[ARP] Automation and Robotics Panel. **Automation and Robotics for the National Space Program.** Administered by California Space Institute, University of California at San Diego. NASA Grant NAGW629, Cal Space Report CSI/85-01. Feb. 25, 1985.

[Bolt] J. David Bolter. **Artificial Intelligence.** Daedalus, Summer 1984, pp. 1-18.

[Camp] J. A. Campbell, ed. **Implementations of Prolog.** Ellis Horwood, 1984.

[Curry] Haskell B. Curry. **Combinatory Logic.** North Holland, 1972.

[DDa] Hubert L. Dreyfus; Stuart E. Dreyfus. Why computers may never think like people. **Technology Review,** 89:1, (Jan. 1986). pp. 42-61.

[DDb] Hubert L. Dreyfus; Stuart E. Dreyfus. **Mind over Machine -- The Power of Human Intuition and Expertise in the Era of the Computer.** Free Press, 1986.

[Firs] Oscar Firschein, et al. **NASA Space Station Automation: AI-based Technology Review.** Administered by SRI International, prepared for NASA-Ames Research Center, Contract No. NAS2-11864. March 1985.

[FM] Edward A. Feigenbaum; Pamela McCorduck. **The Fifth Generation -- Artificial Intelligence and Japan's Computer Challenge to the World.** Addison-Wesley, 1983.

[Gab] Richard P. Gabriel. **Performance and Evaluation of Lisp Systems.** MIT Press, 1985.

[GB] J. Glasgow & R. Browse. Programming languages for artificial intelligence. **Comp. Math. Applic.**, 11:5 (May '85), pp.431-448.

[Gray] Richard M. D. Gray. **Logic, Algebra and Databases.** Ellis Horwood, 1984.

[HC] D. Harel & A. K. Chandra. Horn clause queries and generalizations. **J. Logic Program.**, 2:1 (Apr. 1985), pp. 1-15.

[Haw] F. M. Hawrusik. **Extension of the pebble game for LISP-like programs.** 219 pp. Diss. Abst. Int., Pt B -- Sci. & Eng., vol. 45, no. 2. Aug. 1984.

[Knoe] Arthur Knoebel. **Analysis of High-order Languages for Use on Space Station Application Software, in Research Reports -- 1985 NASA/ASEE Summer Faculty Fellowship Program.** NASA CR-178709, Jan. 1986.

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

[Line] J. Robert Lineback. LISP processor chips point to desktop AI. **Electronics**, March 31, 1986, pp. 17,21.

[Mart] Gary R. Martins. Overselling of expert systems. **Datamation**, 30:18 (Nov. 1984), pp. 76, 78+.

[Mend] Elliott Mendelson. **Introduction to Mathematical Logic**, 2nd ed. Van Nostrand, 1979.

[NAR] Advanced Technology Advisory Committee. **NASA Automation and Robotics -- Information Exchange Workshop Proceedings**, 2 vols. Held at Lyndon B. Johnson Space Center, Houston, Texas, May 13-17, 1985. JSC Artificial and Information Sciences Office, June 5, 1985.

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

[Whit] Daniel E. Whitney. Real robots do need jigs. **Harvard Business Review**, May-June 1986, pp. 110-116.