Four Dimensions of Programming-Language Independence

Daniel J. Salomon

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Technical Report CS-86-13
April 22, 1986

Four Dimensions of Programming-Language Independence

Daniel J. Salomon

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Technical Report CS-86-13
April 22, 1986

ABSTRACT

The properties of programming languages can be evaluated according to how they affect programming-language independence in four dimensions. The four dimensions are: 1) machine independence, 2) problem independence, 3) human independence, and 4) time independence. This paper presents a definition of *independence*, and shows how that definition applies to each of the dimensions. By organizing language features in this way, the strengths and weaknesses of many language designs can be identified, and new directions for programming-language research become apparent. This paper also presents the advantages of independence in these dimensions, occasionally presents the advantages of *dependence*, and proposes methods of achieving independence.

This paper shows that each of the four dimensions can be treated as a discrete domain, and that by classifying the elements of each domain according to their properties, the analysis of independence in these four dimensions is facilitated. The elements of the **machine** domain are classified according to (a) architecture, (b) machine size, (c) peripheral devices, and (d) operating system. The **problem** domain is classified according to (a) discipline, (b) problem context, (c) system mode, and (d) problem-solving methods required. The **human** domain is classified according to (a) user qualifications, (b) natural languages, and (c) the two classes *users* and *implementors*, and (d) independence of the class *implementors* is considered alone. Finally the **time** dimension is treated in three time scales: (a) program processing, (b) project development, and (c) language evolution.

Table of Contents

0. Introduction

Traditionally programming languages have been evaluated and categorized according to a large number of diverse criteria. In this paper we show that these criteria can be organized according to how they affect the independence of a programming language in four dimensions. The four dimensions treated here are

1) machine independence
2) problem independence
3) human independence
4) time independence.

By organizing language-evaluation criteria in this way one can see more clearly how they interact. Furthermore, this organization causes new language evaluation criteria to become apparent.

In addition to showing how the four dimensions encompass existing language evaluation criteria, we show how independence in these dimensions is advantageous and we suggest methods for achieving independence. When a strong case can be made for *dependence* in a particular dimension, it is also presented.

The existing literature on the machine-independence dimension is substantial in quantity, and although the other dimensions are not commonly referred to by the names used here, the literature on them is also extensive. As a result, all the questions raised in discussing these four dimensions cannot be dealt with in full detail. Instead, references are given to survey articles and books, where further discussion and further references can be found, and to user manuals of languages that provide independence in various ways. Only for topics not dealt with in existing literature, or where the approach of existing literature is quite different, is a deeper discussion given.

The technique used here to analyze independence is to treat each dimension as a discrete domain (or a finite set). The machine domain, for instance, consists of all computers. The elements of the domain are then classified into groups according to their properties, and the independence of programming languages on these groups is studied. Again using the machine dimension as an example, the elements of this domain can be classified into groups according to the four characteristics: architecture, machine size, peripheral devices, and operating systems, and programming-language independence of these machine characteristics can then be studied.

0.1. Definition of Independence

In order to analyze the four dimensions, a precise definition of independence is needed. By treating each dimension as a discrete domain and grouping the elements of the domain according to various classifications, a definition of independence can be formulated that consists of two conditions.*

> A programming language can be said to be independent of a classification of the elements of a domain if it:
> (1) supplies the same level of computational power to all groups in the classification, and
> (2) meets the computational needs of each of the groups in the classification.

The first condition insures that the language is neither biased in favour of nor biased against any particular group. The second condition insures that the language is useful for each of the groups. Without the second condition even the empty language could be considered independent. Because of the two conditions in the definition, the word *independence* in this paper is actually used to mean *independence and applicability*. Independence is not treated as a binary property, but rather a language is assigned a degree of independence based on the extent to which it meets these two conditions.

The definition given above can be used to evaluate language features and language-design criteria as well as languages themselves. A language feature can be evaluated according to how it contributes to independence in a language that employs that feature. A language-design criterion can be judged according to whether it implies greater or lesser independence in a language.

* A definition of independence similar to the one presented here was developed independently by Heering and Klint.[29]

1. Dimension 1: Machines

It is not within the scope of this paper to summarize all existing work dealing with machine independence. Fortunately, several survey books have been written where summaries can be found, and these *can* be described here.

Machine independence is often treated as a discipline to be practiced by programmers. Techniques for writing portable programs are summarized by Wallis[54] along with a bibliography of literature on the topic, and by Brown.[8] One would also expect that automatic translators could correct or identify machine incompatibilities. Wolberg[57] and Brown[8] list techniques for such conversions, and Wolberg[57] also lists commercial products available that work for existing languages.

One method of eliminating syntactic differences in the languages accepted by compilers for different machines, and at the same time cutting compiler development costs, is to write a portable compiler. This has led to the techniques of compiler bootstrapping, and retargetable compilers. Ganapathi[17] and Brown[7] present surveys of portable and retargetable compilers.

It has been recognized that a large part of the machine-dependence of a program is due to the operating system under which it is running. To counteract this problem, some have proposed not just a portable compiler but a portable operating system and environment. Such projects are listed in Wallis.[54]

Rather than treating methods for writing portable programs, or methods for writing portable compilers, the focus of this section is on the properties of a language itself that are conducive to machine independence. To aid in this analysis the elements of machine domain can be classified according to the properties:

a) architecture
b) machine size
c) peripheral devices
d) operating system

By studying independence on each of these four properties individually the analysis is simplified.

1.a. Architecture Independence

What are the properties of a programming language that lead to architecture independence? Applying the definition of independence given above to machine architectures would imply that if a language were to be architecture-independent one could:

1) Run any program written for one architecture on any other architecture.
2) Take advantage of all the special features of any particular architecture.

These two goals may seem contradictory, but are not always so. From computing theory one knows that, within the bounds of memory size and computational speed, all digital computers that can implement a universal Turing machine, have equivalent computational capabilities. It follows that any special feature of one architecture can be simulated on any other architecture, and in many cases this simulation can be accomplished with no significant penalty in speed. It is therefore possible, with the language designer and the language implementor working cooperatively, to achieve architecture independence.

Consider, for instance, the operators increment "++" and decrement "--" in the C language. They were designed to take advantage of the autoincrement and autodecrement addressing modes of the PDP-11 computer. Using these addressing modes can speed up a program by reducing the instruction count. Although they were intended for the PDP-11, the increment and decrement operators can be easily implemented on other computers, even though those computers do not have these special addressing modes, and the implementation can be just as efficient as for any other language construct that performs incrementation.

Why would one want a language to have both properties of independence? The desirability of the first property is well recognized. It implies the portability of programs across architectures and thus permits programmers to share algorithms, software vendors to access a larger market, and users to change or mix manufacturers.

The second property is desirable because of the realities of computer marketing. A manufacturer can sell his computer only if it is better in some way than his competitor's, or if it is cheaper. If it *is* different, the customer must be able to make use of that difference, that is, he must be using a programming language that can take advantage of that difference. The major sources of compilers today are the computer manufacturers and software houses under contract to computer manufacturers, and a computer manufacturer will not in general be willing to finance a compiler project that does not make use of the features of his architecture that make it superior to his competitor's architecture.

There are many possible architectural variations. They include such things as operand alignment, address size restrictions, special address modes, cache or memory page sizes, the varieties of data types and their properties, character codes and their collating sequence, and complex microprogrammed instructions. With all these variations, it is unreasonable to expect programs to have the same space and time characteristics on all machines. Similarly, variations in word sizes and floating point formats make it unlikely that programs will produce the same results on all machines. Establishing strict standards for machine design would be equivalent to abolishing architectural variations, and as explained above, this is undesirable. Similarly, emulation of a standard machine on all target machines would be a rejection of architectural variation and could be very costly. The language designer should therefore not expect perfect portability, but rather should be satisfied with producing acceptable results on most machines.

Surprisingly, major architectural properties such as **parallelism** need not affect a language design. For instance, the writers of the CRAY FORTRAN compiler decided that the compiler could do an adequate job of discovering parallelism automatically in unenhanced FORTRAN for their machine.* In the opposite vein, the languages FP[6] and Lucid[52,53] demonstrate that it is possible to design languages for parallel execution that are also practical on sequential machines. They do however require programmers to change substantially their program-design techniques.

1.b. Machine-Size Independence

Another way to classify machines is by machine size. Machine size can be independent of architecture since the same architecture can come in different sizes of machine.

Computers have been getting more powerful, that is, they have been getting faster, and their memories have been getting larger. Nevertheless, new uses have been found for the less-powerful machines since they have been getting physically smaller and cheaper. Machines equivalent to the 8K minicomputers that once commonly appeared in small laboratories, and strained a programmer's skills, now appear in automobiles and household appliances. In addition, the consumer computer industry markets a wide range of computers from hand-held and lap-top computers to small office systems. Thus programming techniques for small computers have not disappeared; they have merely found new applications. Language designers accustomed to working on large mainframes should not abandon their poor cousins who must work on small machines, and force them to use archaic languages like BASIC. Furthermore, a large computer often has several small computers attached which handle peripheral functions, and networks of computers of different sizes are becoming more common. Because of this, a programmer may have to deal with many different sizes of computers in the same day, and it would be advantageous if he could do so in the same language.

Paraphrasing the definition of independence, one could say that a programming language that is independent of machine size has two properties:

1) Programs written for one size of machine are, as far as is practical, able to run on the other machine sizes.
2) The particular requirements of each machine size are met.

These are not easily achievable objectives. If one looks at existing programming languages, one sees that perhaps the most significant differences are between those intended for use on small machines and those intended for use on large machines. PL/I, for instance, is intended for large machines and would be very difficult to implement with small machines as either the host or the target machine.

* The problem of discovering and scheduling parallelism is NP-complete in the general case.[19]

FORTH, on the other hand, was designed for small machines and could easily run on large machines, but since it lacks the features that large-machine users have grown accustomed to, it is hardly used there.

There are certain language properties that reduce a language's suitability for use on small machines:

- dynamic memory allocation and garbage collection
- a large number of standard types and operators
- a large library of standard functions
- thorough error checking
- a non-regular syntax

Any of these properties alone can be implemented on small machines, but taken together they become unmanageable.

Just as users of large machines demand a rich language, users of small machines demand special properties of a language too. The compiler must be small and fast, and the code generated, including the run-time library, must be compact and fast, therefore the language must permit the generation of simple efficient programs.

Interpreters are often preferred over compilers when the development machine is small, because an interpreter can usually be made smaller than a compiler. In addition, an interpreter can provide a complete development environment, in which the user can quickly edit and test his code without the time consuming operations of loading and running a separate editor, compiler and linker.

1.b.1. Achieving Machine-Size Independence

The needs of large machines (a rich language) and the needs of small machines (a small, fast language) are hard to reconcile. One way to meet these requirements is to organize a language as a series of two or more dialects, growing in complexity to meet the needs of larger machines. The smaller dialects would be proper subsets of the larger dialects, thus achieving independence with increasing machine size.

Independence of decreasing machine size could be provided by translators from the larger dialects to the smaller ones. To permit the writing of such translators, the smaller dialect must be as suitable a target for automatic translation as is the assembler language of the larger machine. Because of the limits on physical memory and on the speed of small machines, not all programs written in the larger dialect would be useful on small machines, but for those that are useful, downward independence of machine size will have been achieved.

The use of dialects suitable for different machine sizes would permit the development of programs on a large, comfortable host system in a full powered language, followed by translation for a small target system. Alternatively, the programmer could work directly on the small target machine and pass his large language through a series of translators until machine code for his target results. In this case, each one of these translators must be small enough to run on his target machine.

A language organized in versions of increasing complexity would also be easy to implement from scratch. The smaller versions could be implemented first, which could then be used to bootstrap the larger more complex versions.

1.c. Peripheral-Device Independence

One can categorize computer systems according to their peripheral devices, and then analyze programming-language independence on these categories. A programming language should be able to make use of all the peripherals attached to a machine, otherwise some other programming language will be needed to fill the gaps.

Traditionally, peripheral independence has been handled by dividing peripheral devices into classes and providing language constructs to handle those classes. There are usually at least two classes: sequential devices and direct-access devices.

The ultimate device independence would be achieved by making all peripheral devices look exactly like memory. Since peripheral devices are simply storers or suppliers of information, why not handle them all as memory?

Virtual memory is an example of this concept, and programs running on a virtual-memory system constantly access secondary storage, even though they contain no I/O instructions. Other examples can be drawn from the Multics operating system,[45] which has a unified syntax for accessing variables and files, and from the concept of permanent variables,[11] variables that keep their value between runs of a program, presumably because those variables are actually in secondary storage.

1.c.1. The Benefits of Device Dependence

Although theoretically all I/O statements could be eliminated from a programming language, they continue to exist because they usually allow the programmer to handle I/O devices more efficiently than if he relied totally on automatically generated code. Since I/O operations usually require execution times that are orders of magnitude greater than those required for computation, efficient handling of I/O is an important consideration.

Peripheral dependence can have even greater benefits for a language. Consider for instance the language APL.[22] Because it relies fundamentally on a special character set, it requires special peripherals. The special shapes of the characters and the systematic way that operators are overprinted to generate new operators, greatly assist the programmer in visualizing the highly complex and rich operators of APL. Attempts to map the APL character set into ASCII have been largely unsuccessful because they destroy the connection between the shape and the meaning of the operator. Modcap[56] is another example of a language that makes good use of non-ASCII characters.

Modcap[56] is another example of a language that makes good use of The character set conundrum afflicts almost all programming language designers—how to assign the rather limited set of ASCII special characters to denote a rich set of programming language operators. The matching of some characters and operators is obvious and easy: "+" for addition, "–" for subtraction, "/" for division, etc. Usually the first great decision comes when one must chose a character for use as a comment delimiter. The symbols "$", "¢", "/* ... */", and "#" have all been used by various languages, simply because they were left over after all the meaningful operators had been assigned, and not because they represent the concept of a comment in any way.

One solution to this language design problem would be to select ASCII characters where they were suitable, and to design new characters when no suitable ASCII character exists. To permit peripheral independence, select a suitable unique ASCII character sequence to represent the invented character. It is important to chose substitute character sequences that can be automatically translated to their preferred representation. By this technique the language can be used on existing peripherals, but should the new language become popular, special peripherals for use by this language that support its special characters would become available as they have for APL. Remember that the APL character set was designed at a time when designing new characters also required the design of new hardware, but now, with laser printers and bit-mapped displays, new character sets can be designed in firmware or software.

A good illustration of the above technique is given by the use of braces "{ ... }" in Pascal to delimit comments. Braces do carry the connotation of a parenthetical remark, and therefore their form represents their function. Braces, however, are not available in the EBCDIC character set and so they are replaced by the less convenient sequence "(* ... *)" on peripherals using that character set.

There are other examples of the benefits of peripheral dependence in a programming language. LOGO,[4] for example depends on the existence of a turtle-graphics device. This dependence yields a simpler graphics language than if one tried to handle all possible graphics devices. As another example, if an associative-storage device were invented, its full benefits could probably not be realized if it were treated like any other storage device.

The principle conclusions of this section are that a truly device-independent language would treat all devices as memory, and that a language designer should feel free to invent new characters for his language if they clarify the concepts.

1.d. Operating-System Independence

Machines can be categorized according to what operating system they use. The operating system in use can be treated as a machine characteristic because, to a programming language, the operating system appears to be an extension of the hardware. It supplies the device drivers that make the peripherals look alike, and in many cases, parts of the operating system are coded in firmware (especially on virtual-memory machines).

Interaction between a program and the operating system is usually carried out via procedure calls. In order to allow this interaction, the language designer or implementor must provide for the invocation of operating-system procedures. This means that he must adopt the standard procedure-invocation conventions for his language's procedures, or provide a special interface to the standard conventions. If he fails to do this, his language may meet the first condition of independence—uniform computational ability across operating systems—but would fail on the second—meeting the special needs of an operating system. A user of such a language would be handicapped in his control over the file system and over other tasks, and his use of system information, such as user identification and execution times. This handicap would relegate the language to second-class status.

Once a programmer uses system-specific procedures, however, his code will be non-portable. Thus a portable language should include the definition of a set of standard procedures for the common operations that require operating system intervention. It is impossible to foresee all the facilities that operating systems could provide, but the more that the language has built in, the less likely it is that a programmer will have to use system-specific procedures. If a language, however, has all possible operating system calls built in, then the effect would be the same as the technique of writing an entire portable operating system.

2. Dimension 2: Problems

Programming languages are used to solve problems. The greater the *problem independence* of a programming language, the larger the number of different types of problems it can be used to solve. To help in the analysis of independence in this dimension, problems are categorized here according to four different properties:

a) discipline or application
b) problem context
c) system mode
d) problem-solving methods required

2.a. Discipline or Application Independence

There is a long tradition of application-specific programming languages. For example, COBOL is intended for business applications, FORTRAN for scientific applications, C for systems programming, and LISP is considered to be an AI language. Because of this tradition, there is a common belief that certain applications have peculiar needs that cannot be met by a common language. If, however, one examines the capabilities of these various languages one will see that there is a great deal of similarity in the data types and control structures offered*. Furthermore, if one examines a language designed for a particular application, one will find that any unique feature that it has would sometimes be useful to programmers in other applications, and any feature it is missing would sometimes be useful in the language's intended application. Indeed, since many programmers work in more than one discipline, and a single program itself might span more than one discipline, it seems foolish to design a language intended only for a single discipline.

* Even in LISP, which is a functional language, it is common to use functions that mimic the control structures of algorithmic languages.

Why then are there application-specific languages? The reason seems to be that it is easier to design a language that handles only the most common needs of a particular discipline, and handles those only in their most common form, than it is to design a programming language that meets the needs of all disciplines in all forms.

Two approaches have been taken in the past toward designing an application-independent language. The first is the PL/I approach. This approach is to include, nearly unchanged, most of the features of two or more application-specific programming languages. In the case of PL/I it was the merging of COBOL and FORTRAN, and the inclusion of features from other languages such as ALGOL 60. This approach leads to a very complex language with considerable redundancy, and a large expensive compiler. A programmer working exclusively in a particular discipline will commonly use only those features that he needs for his particular application, but has the option of using other features should the need arise. The problem with such a language is in designing it so that the features taken from different languages do not collide, and in the enormous cost of implementing, maintaining and using such a compiler.

The second approach is the one taken by the designers of ALGOL-68,[51] Ada[1] and other languages. That approach is to try to generalize control structures and data types so that they meet the needs of all disciplines, but are assembled in a uniform and orthogonal way. The approach generally produces a simpler language with a smaller faster compiler than the previous approach. One problem with this approach is the extra effort needed to discover the most general form of a concept so that it can provide the functionality of several simple features. Another problem is that a programmer will not always easily see how a generalized concept can be applied to his particular problem.

Looking specifically at Ada one can see the current state of the art in the second approach. To meet the needs of the many different applications, it uses the following strategies.

- All control is handled by a small set of well recognized control structures.
- The set of predefined data types is small, but a well recognized set of type constructors permits the creation of new types as needed.
- It permits the overloading of existing operators to handle the new data types in a natural way.
- Procedures intended for a particular application alone are grouped in packages which can be included or excluded at will. This grouping avoids burdening other applications with any added complexity.

The last point in this strategy is the most demanding. If procedures are to supply the special needs of all disciplines, then the form of procedure invocation must be very flexible and powerful. One should be able to pass procedural parameters, a variable number of parameters, arrays of variable size, and parameters of variable type. One should also be able to test the characteristics of the actual parameters passed. Ada meets most of these requirements, and the requirements that Ada does not meet are the subject of on-going study, including such things as parameter-type polymorphism,[27] and execution-time manipulation of data types.[14]

Ada has been attacked for being too large a language,[32] but this size is not directly the result of its attempts to be application-independent. Has the Ada approach lead to a language that is smaller than PL/I? It is actually not easy to compare the size of two languages accurately. One cannot compare the sizes of the language manuals, since manual size varies considerably with writing style, accuracy, and completeness. Nor can one compare the sizes of the compilers, as these vary with programming style, efficiency and quality of generated code. Not even the size of the grammars can be compared, since grammars are often written in a redundant fashion to simplify compilation. Furthermore, none of these measures of size consider the extra functionality that one language may provide over another. As a result this question remains open to debate, but it seems that the Ada approach should lead to programs that require fewer language constructs than equivalent PL/I programs.

2.b. Problem-Context Independence (Orthogonality)

When one writes a program to solve a problem one must usually break that problem down into subproblems or steps. Sometimes the subproblems have a great deal in common, but appear in different contexts. One may, for instance, wish to evaluate an arithmetic expression in an assignment statement, a loop control statement, or an output statement. The classification of the problem domain treated in this section divides subproblems according to the context in which they appear. If a programming language has independence of problem context then it provides the same facilities for solving subproblems regardless of their context, and it meets the needs of the various contexts. In the literature this language property is commonly called *orthogonality*—the ability to apply language constructs independent of each other.

Examples of lack of orthogonality are easy to find. ANS Pascal,[35] for instance, has a rich set of expression operators, but in constant expressions only the negation operator is allowed. FORTRAN, for another example, has different looping constructs for program flow (DO loops) and I/O statements (implied DO loops). On the other hand, orthogonality was one of the principle design goals of ALGOL-68, and that language has a high degree of orthogonality.

The merits and disadvantages of orthogonality are covered extensively in texts on programming languages (see for instance Ghezzi and Jazayeri[21]). The principle advantages are that it simplifies the description of the language syntax, makes learning the language easier, and provides great power in all contexts. The disadvantages are that the meaning of particular language constructs applied in some contexts can be hard to establish, and that the implementor must handle many strange combinations of constructs.

2.c. System-Mode Independence

Another way that one can classify problems is according to the *system mode* that they run in. Most operating systems have more than one of the following modes.

1) operating-system command mode
2) text-editing mode
3) application mode (user-written programs)
4) data-entry mode
5) preprocessor mode*
6) text-formatting mode
7) any other system utility that accepts a command language.

The trend in operating systems has been to give each of these modes more and more algorithmic control. Thus command languages, editor macro languages, preprocessor languages, text formatters, and the other modes have been gradually endowed with more and more data types and control structures. The data types, operators, and control structures usually come from a standard set including:

a) character strings and operators
b) numeric types and operators
c) conditional execution
d) case selection
e) looping
f) procedure invocation†
g) parallel execution
h) backtracking.

Although the algorithmic control structures of the various system modes have a great deal in common, they often vary in particulars of syntax and semantics. If a programmer is to make full use of his operating system he will be forced to learn five or six programming languages, one for each mode, and to

* Preprocessor mode is included in this list for completeness, but is dealt with in more detail in the section on time independence under the time scale *program processing*.

† Many of the features in this list as well as esoteric features like string pattern matching, associative arrays, and dynamic memory allocation, can be implemented via procedure invocation.

switch repeatedly from one language to another in the same session.

Can a single mode-independent programming language be designed that will give all these modes the same computational power, and meet the special needs of the different modes? Some existing languages have been reasonably successful in unifying some of the different modes. These include BASIC, SMALLTALK,[23] INTERLISP,[49, 39] and work in progress by Heering and Klint.[29]

2.c.1. The Properties of a Mode-Independent Language

What properties should a system-mode-independent language have? No matter what mode a program is intended for, the arguments in favour of structuredness, safety, readability, etc., apply for all programs that are stored in permanent files. The language should be interpretable, since most system modes interpret their code and give immediate results. The language should also be compilable to provide efficiency in the application mode. A compilable language would also benefit other modes, since popular editor macros and command-language procedures could then be compiled into efficient operations without having to be rewritten in another language.

The syntax and semantics of a mode-independent language should be identical in all the modes, since a language that changes subtly in meaning in various modes may be worse than totally different languages. One way to achieve this uniformity would be to use a single system-resident, re-entrant parser-interpreter, and have all of the modes invoke that parser for command input. Each active mode (modes may be nested) would then have its own separate symbol and value tables, but would use the same parser.

The standard control structures presented above should be adequate for all modes; the main differences between modes appear in the *commands* available. Commands can be thought of as procedure calls in a standard programming language and can be implemented as such. To meet the needs of procedure calls in the various modes the language should have a very flexible procedure call statement. Some of the properties needed in a procedure call are:

a) positional parameters (with a variable parameter count)
b) keyword parameters (with default values if omitted)
c) option parameters (their presence selects an option).

If a system were designed in this way—a central parser with a special set of procedures for each mode—there would be more benefits than just a uniform programming language. The whole operating system would be smaller, since each mode would not need to contain its own parser. Although the system would be harder to design, it would require less time and effort to code. A language-specific editor would become a fruitful project since it would be useful for many different types of files. Such an editor could do syntax checking during program entry, provide indentation to match program structure, and do semantic pattern matching, rather than just string pattern matching. (See for instance Teitelbaum and Reps.[48]) Finally, improvements in efficiency of the central parser-translator, or in the efficiency of the code it produced, would benefit all modes.

There is at least one serious incompatibility between the desired properties of languages for the different modes. It is ease of entry. Some modes, such as command mode and editor mode require very short command names (procedure names) and a very concise syntax so that a user can easily enter those commands interactively. Most algorithmic languages, on the other hand, use verbose constructs and strict type-declaration rules to provide a measure of error resistance. The modes that require a concise language would benefit from the error resistance of a verbose language once an algorithm was entered into a permanent file, but the problem of quick entry of interactive commands remains.

The problem could be handled by an input filter that translates a concise input form into an error-resistant permanent form. The conversions it performs could include: keyword-abbreviation expansion, automatic default declaration of variables (with interactive user approval), and automatic quoting of string constants.

2.d. Independence of Problem-Solving Methods

Many standard methods for the solution of specific types of problems have gained widespread acceptance. They are popular either for their simplicity, efficiency, reliability or clarity. In order for a programming language to be problem-independent, it should permit the use of those methods. The following is a cursory list of the method-specific features generally expected in a high-level language:

arithmetic operations
real arithmetic
complex arithmetic
trigonometric and mathematical functions
plotting and graphics
recursion
linear arrays
multidimensional arrays
conformant arrays
records
pointers
dynamic memory allocation
linked lists
stacks
queues
trees
graphs
hash tables
elapsed run-time measurement
real-time control
parallel processing.

A methods-independent language should provide for the convenient use of these methods. Some languages have some of these methods built into their definition. LISP, for instance, has built in list structures, and SNOBOL has built in hash tables. The more common approach, however, is to provide abstract language features that can be used to implement these methods. To use this approach a language requires flexible type constructors and flexible procedure invocation. By providing the language features at a more abstract level, a language will not be tied to supporting an obsolete problem-solving method and can adapt to new ones.

2.d.1. Traditional Mathematical Problem-Solving Methods

The problem-solving methods listed above are the ones that are commonly used in programming projects. There are, however, traditional problem-solving methods that are commonly used in mathematics, but less commonly by programs. These methods are the ones of algebra and calculus in which expressions are manipulated rather than numeric quantities. When a programming language supports these methods they are called *symbolic computation.*

There are languages such as MACSYMA[2] and Maple[9] that are designed specifically for symbolic computation. These languages are oriented toward a specific problem-solving method, and although they can be used as general purpose languages they are not the optimal choice for most applications. It is possible to provide symbolic manipulation in a general-purpose language by the use of a package of symbolic-manipulation procedures, but a language preprocessor such as ALTRAN[25] for FORTRAN or FORMAC[58] for PL/I is usually needed to make such a package convenient to use. In a language that supports abstract data types and operator overloading, such as Ada, symbolic expressions and their manipulations could be described in a notation that closely resembles that of MACSYMA or Maple, but of course an amount of work, equivalent to that expended in developing the expression-manipulation libraries of those languages, would still be required.

In the physical sciences it is a common practice to associate units of measure with variables and constants, and to carry these units along in computations. Without units of measure the values in a calculation are meaningless, since a distance of "5" could mean five microns or five light years. In addition, the carrying of units during calculations helps to verify the correctness of the manipulations being performed. In most programs, the units of the quantities being manipulated are implied, not explicit, and errors in units are a significant source of programming errors. Some work has been done by House,[34] Gehani,[20] and Männer[40] to analyze the possibility of including units of measure in programming languages. Such a feature is called *dimensional analysis*. Dimensional analysis can be supported at run time by the definition of a suitable data structure, and of a package of procedures for performing calculations with values and their units. The ability to overload operators assists in implementing dimensional analysis so that it can be used in a natural way. Indications are, however, that compile time checking of units of measure is not feasible without extensions to a language specifically for this purpose.[40]

2.d.2. Other Method-Dependent Languages

There is a problem-solving method called functional programming that can significantly affect the design of a programming language. LISP and FP[6] are the languages best known for this property. A purely functional programming language is most easily recognized by what it does not have, rather than by what it has. It has no variables nor side effects. Despite the considerable attention the functional programming languages have attracted, it is not necessary to have a language designed specifically for functional programming. A programmer can easily use a functional style of programming in any language that provides for the definition and recursive use of functional procedures, and he can easily be provided with an automatic program checker that locates violations of the rules of functional programming.

Another problem-solving method that has attracted a lot of attention is *declarative programming*. Examples of languages designed with this method in mind are PROLOG[10] and Lucid.[52, 53] It is the goal of some current general-purpose language design projects to be able to implement this problem-solving method too, using a suitable data structure, and a package of procedures, but no well-known examples of such packages exist as yet. Extensible distfix operators[36] may be needed to provide constructs equivalent to those of Prolog and LUCID in a natural way.

The conclusion of this section is that method-specific programming languages may eventually be replaced by general-purpose languages. With suitably flexible data-type constructors, operator overloading, and the preparation of an appropriate package of procedures, a general-purpose programming language should be able to handle a broad range of problem-solving methods in a natural way.

3. Dimension 3: Humans

In studying the human dimension, four ways of classifying human computer users are presented:

a) user qualifications
b) natural language used
c) users and implementors
d) implementors alone.

In all these classifications a common problem exists: our incomplete knowledge of human psychology and human behavior. It is almost impossible to make a general statement about human preferences or behavior, and therefore most statements must be highly qualified. It is important, however, not to neglect humans when designing programming languages, since the principal function of a programming language is to permit communication between humans and computers.

3.a. User Qualifications

Computer users can be classified according to their qualifications or competency.

Four classifications of user competency are discussed here:

1) students (and educators)
2) casual or occasional programmers
3) professionals
4) experts.

One interesting property of this classification scheme is that the membership of each group changes continually. At one time all programmers begin as students, and then gradually progress to the other groups. Even an expert programmer may become a student again should he decide to learn a new language that is radically different from those he knows. Furthermore, the group members are different for different languages, since an expert in one language may be a casual programmer in another, and just a student of yet another language.

If a programming language is designed solely for one of these groups, then it assumes the existence of a more advanced language to which a programmer can graduate as his expertise improves, or of a more primitive language with which he can be introduced to computing. In the case where a language excludes the needs of students, the designers should name which language or languages are suitable stepping stones to his language. Nevertheless, de Remer and Kron[13] have given arguments for targeting a language at a specific group, and there are many examples of languages targetted specifically at students (LOGO,[24] BASIC, etc.) and of languages targetted at experts (C, ALGOL-68, etc.)

3.a.1. Achieving User-Qualification Independence

Let us assume that a language independent of user qualifications is desirable and explore how it could be attained.

The first criterion for independence—providing the same level of computational power—is easily achieved by giving all of the above groups access to the same translators, compilers, and program preparation tools. So let us turn our attention to enumerating the special needs of each group.

All of the groups need good documentation but on different levels. Students need introductory texts and the other groups need reference manuals. The professional and expert programmers may also need in-depth reference manuals that cover every possible aspect of the language and of the implementation of the language that they are using. Introductory texts and reference manuals can be written for any programming language, but certain properties of the language can make the job easier. To facilitate the writing of introductory texts the language should have a small subset that is adequate for short introductory programs. The text should not have to excuse itself with such phrases as, "Never mind the *such-and-such* statement; it will be explained later." When a tremendously powerful construct leads to confusing programs for simple problems, then a redundant simple construct should also be provided. In FORTRAN for instance, free-format I/O greatly simplifies the job of writing simple programs.

For a programmer to develop a good coding standard on his own takes years of experience. As a result, for the benefit of novice and occasional programmers, a coding standard should be part of the language description. This would not only reduce errors, but also make it easier to read code from different shops.

The reference manuals as well as introductory texts would be simpler and shorter if the language is uniform and orthogonal. It also helps if the language is small, but it takes a great deal of effort to design a language that is both small and meets all the needs of expert programmers.

Many of the needs of these four groups are actually requirements placed on the compiler implementor, rather than on the language designer. These requirements include such things as good run-time error checking, clear error messages, fast compilation, and efficient generated code.

Efficient generated code is of greatest importance to professional programmers. It can determine whether or not a project is feasible (in time or cost) and can determine the market share of his product. It is easy to write a slow, poor compiler for any language, but sometimes the design of a language can

prevent the writing of efficient, good compilers.

Beginners and occasional programmers who often write run-once programs are more concerned with the time taken to write and debug a program than they are with the efficiency of the final code produced. Because of this, a language designer must be careful to balance his design and reach some optimal program-preparation-time versus run-time tradeoff. Often the nature of the tradeoff is hard to determine. A strongly typed language, for instance, will usually require more coding time, since the programmer will spend extra time preparing declarations, but occasionally will save long debugging periods that greatly exceed the coding time. No precise way exists of predicting the exact benefit of such features and thus the language designer must often make his decision based on incomplete information.

3.b. Independence of Natural Language

The elements of the human domain can also be classified according to the natural language that they speak. So far, the natural language of the programmer has largely been ignored by language designers, and almost all existing programming languages use the Roman alphabet (without the accents used by the romance languages) and English keywords. This oversight is largely due to the fact that most early research in computer science took place in English in Great Britain and the United States. However, it is inevitable that as computer power reaches a larger and larger proportion of the earths population, programming languages will have to become more accessible to non-English speakers. Also, there are many multinational companies, such as IBM, ITT, and Honeywell-Bull, that have programming shops in different countries. Sharing programs and documentation can be a problem for them.

Applying the definition of independence to this classification of the human domain, one can conclude that a language does not provide the same power to different natural language groups unless it allows the members of each group to use their own language for comments and for mnemonic symbols, such as keywords and variables. They also require introductory texts and reference manuals in their own language. By allowing the members of the different groups to use their own language for programming, one would also be satisfying the second condition for independence, meeting the special needs of each group. Currently, the problem of providing computing facilities to members of other linguistic groups is achieved by forcing them to learn English, which is in effect moving all computer users into the same linguistic group.

Many contend that the keywords, syntax and semantics of a programming language take on a meaning of their own that is distinct from the natural language of their origin, and hence the keywords chosen are irrelevant. As a result, they justify the invention of new keywords such as *esac* and *pragma*, (see Eastman[15] for a discussion of this topic.) Few, however, would argue that the use of mnemonic keywords is very helpful, at least in the learning stages. Furthermore, for some language groups, such as the Japanese, the Russians and the Arabs, the difficulty of learning keywords in a foreign language can be compounded by the problem of learning them in a foreign alphabet. Conversely some non-English programmers say that by using keywords in English and variables in their own language, they have a convenient mechanism for distinguishing the two that is almost as effective as using boldface keywords. The distinction would be even more apparent if the keywords were in a different alphabet.

Is it possible to design a programming language that is independent of natural languages? The programming language that comes closest to this goal is APL. It has no keywords and its operators are invented symbols independent of any natural language or alphabet. APL does depend on the Arabic numerals, but these are now almost universal.

It would not be difficult to adapt APL so that any alphabet could be used for variables. With the knowledge that all alphabetic strings are variable names, any APL programmer could read and understand any such program regardless of his linguistic group. The meaning of comments and the mnemonic worth of variable names would of course be lost, but the algorithm expressed would not be ambiguous.

The same level of independence of natural language achieved by APL could be achieved for any programming language by the use of automatic translators. The alphabet and keywords of a Japanese compiler, for instance, could be automatically translated into a form acceptable by an English compiler. Programs could be translated so that the algorithms were 100% identical, but of course, as with APL, the mnemonic worth of identifiers and the meaning of comments would be changed or lost without the

intervention of a knowledgeable human translator. As a result, the programming language would be independent of natural language for communicating algorithms between a human and a computer, but not for communicating algorithms between two humans.

3.c. Users and Implementors

Humans can be divided into two important classes: programming-language users and programming-language implementors (these groups are not necessarily disjoint). The users want a language that is easily usable, and the implementors want a language that is easily implementable. When a language designer gives his language a feature or property, he should determine whether the purpose of that feature is to make the language easier to use or easier to implement. In every case he should determine if the cost or benefit to the user balances the benefit or cost to the implementor.

There are many examples of clearly implementor-oriented features in existing languages.

- Many versions of BASIC require that identifiers be no longer than two characters and FORTRAN 66 has a limit of six characters.
- Pascal requires that the sizes of all array types be compile-time constants, and does not provide a convenient method of initializing arrays.
- The postfix notation of FORTH simplifies the interpreter but is widely recognized as difficult to use.
- The single character operators of TECO provide substantial computational power, but result in unreadable code.
- LISP, from the very start, was designed to be implementor-oriented.[42] The obvious examples of this are the choice of the function names `CAR` and `CDR`, and the full parenthesization of expressions required by prefix notation. There are many more good examples, but they are also more controversial.

It is also fairly easy to list user-oriented features.

- The requirement in Pascal and other languages that the type of all variables be declared before use is designed to reduce user errors. This is an example of a feature that is a restriction on the language, but is, all the same, user-oriented.
- Automatic garbage collection makes programming considerably easier, but can be difficult to implement. This is one feature of LISP that is definitely user-oriented, and a feature notably missing from some versions of Pascal.[35]

The remaining features in this list are rarely seen in programming languages.

- When writing large numbers, digit separators, such as commas or spaces, help humans read the number accurately. They would be helpful in source code, input data, and output data, but no common language permits digit separators in all three cases.
- Programmers use indentation to identify the structure in their programs. Most compilers disregard this indentation, and recognize only keyword and punctuation delimiters, because indentation is hard to parse. (A parsing strategy for indentation has been proposed by Leinbaugh.)[38]
- Statement separators, such as semicolons, simplify the implementor's job, but users easily forget them, which can result in hard-to-find bugs. An end-of-line, in conjunction with explicit statement continuation marks, would be a more user-oriented statement terminator, but harder to parse. To eliminate all ambiguity, explicit statement continuation marks should be placed both at the end of the continued line, and at the beginning of its continuation. This method has been proposed for the next FORTRAN standard.[3]
- Redundancy is a common error prevention strategy in everyday life, but it seems that language designers try to find the minimum syntax necessary to describe an algorithm.

Implementor-oriented features are not in themselves bad, nor are user-oriented features always good. Adding implementor-oriented features to a language or leaving out some user-oriented features, can make the difference between a reasonably usable language whose compiler is delivered on time at a

reasonable cost, and a highly usable language that goes over budget or is never delivered.

3.c.1. Users as Implementors

A program is both a data manipulator and data to be manipulated. It is data not only for compilers and interpreters, but also for text editors, pretty printers, cross-reference generators, and all sorts of other minor program manipulators. It is also output data from program generators, program formatters and program translators (such as the language-version translators proposed in this paper). If one considers anyone who has worked on one of these types of program processors to be an implementor, then the class *implementor* is very large.

Few people realize how often they write program-manipulation routines. If, for instance, a user types editor commands that rename every occurrence of a particular variable in a program, then he has written a program manipulator. Even in the simpler case where one needs to find all print statements in a program using editor commands, one has written part of a program recognizer. Such manipulators need to be composed frequently and quickly, and therefore present an argument for an extremely simple and redundant syntax in a language. If a grammar for a language were extremely simple then user-written program manipulators would be even more common.

There are, however, many examples of what can happen when a programming language is made too simple. Many text-editor macro languages and text-formatting languages use single-letter or double-letter keywords and variables, in order that they can be interpreted quickly and easily. As a result, they are easy languages to manipulate. But often, also in order to keep them simple, they are not given enough computational power to be generally useful, or they are too cryptic to represent an algorithm clearly.

3.d. Implementor Independence

In this section, instead of considering the entire human domain, an important subset of humans—the implementors—will be treated alone. Because implementors can have a significant effect on the language accepted by their compiler*, such a study can be very rewarding.

Rephrasing the definition of independence given in section zero, one could say that a programming language is implementor-independent (or implementation-independent) if:

1) Any program accepted by any implementation is accepted by all implementations.
2) The language meets the special needs of the customers of any particular implementor.

The customary method of meeting the first condition of implementor independence is by providing formal specifications for the syntax and semantics of the programming language, and to prepare a validation set of programs to test the final products of implementors. The formal specification of syntax has reached quite an advanced state (see for instance Harrison[28] or Aho and Ullman[5]). Techniques for formal specification of semantics are discussed by Marcotty et al.,[41] Hoare,[31] Tennent,[50] and Wegner.[55]

The second condition of independence, meeting the special needs of the implementors customers, may seem to be at odds with the first condition, but in fact, most of the special needs of a customer can be met without changing the source language. These needs are such things as:

1) a fast compiler,
2) high quality code,
3) code for a specific machine,
4) extensive error checking, and
5) clear error messages.

There are times, however, when a customer may truly need a slight or major modification to the language. This is due either to the fact that a language designer cannot foresee every possible use that will be made of his language, or to the fact that he may have made inappropriate tradeoff decisions in the design.

---

* For simplicity, here as elsewhere the word *compiler* is used to represent compilers, interpreters, and translators.

There is a long tradition in computer science of implementors making enhancements to, or placing restrictions on, the languages they are implementing, and one can expect this tradition to continue. In fact, the existence of implementor modifications to programming languages has had a beneficial effect on many languages. Most of the improvements of FORTRAN-77 over FORTRAN-66 were inspired by implementor modifications, and had already been tried and tested. Only a short-sighted language designer would claim that his language was perfect and complete for all time. Allowing implementor modification would give a more realistic designer a larger source of ideas for his next release of the language.

The chaos that could result from allowing implementor modifications to a language can be minimized by placing restrictions on the type of modifications that can be made, and the way that they are made. Such restrictions could be that:

1) Additions to a language should follow the spirit of the original language, if the language has an easily recognizable or documented philosophy.
2) Additions to a language should be automatically recognizable as extensions to the language by compilers of the unenhanced language.
3) Deletions from a language should be automatically recognizable by a compiler for the modified language.
4) If possible, implementors of a modified language should provide automatic translators that can translate the new language into the standard language and vice-versa.

4. Dimension 4: Time

The time dimension is considered here in three different time scales:

a) program processing,
b) project development, and
c) language evolution.

Although one would expect the time dimension to be a continuous domain, will be seen that this dimension also, in all three time scales, is discrete.

4.a. The Program-Processing Time Scale

There are commonly as many as five phases of computation during the processing of a program. They are:

1) preprocess
2) compile
3) link
4) load
5) execute

As processing proceeds through these five phases new information becomes known and different types of computation need to be done. Let us examine the kind of information and computation found in each phase.

In the preprocessor phase the information supplied concerns properties of the program source, and the algorithms are mainly ones that will modify the parse of the program during the compile phase. In the compile phase, literal constants and the programmer's run-time algorithm become known. Traditionally, the only compile time computations that a user can control are expressions involving constants. In the link phase and load phase new information becomes available about relative and absolute addresses, and about the external and system procedures to be employed in the execution phase. Traditionally the user has little or no control over computations in these two phases, all computations being specified by the compiler, the linker or the loader. In the execution phase, all remaining information about the problem

to be solved becomes known in the form of the input data.

In order for a programming language to be independent of these five phases, it would have to supply the same computational power to all five phases and also meet their special needs. Let us explore what has been done and what could be done for each of these five phases to achieve independence.

In most programming systems, when a preprocessor pass is available it is in a language significantly different from the other phases. The preprocessor language is most commonly a purely string handling language, and usually this suffices. Nevertheless, instances arise when a full-featured preprocessor language, that includes numeric calculation and I/O, would be valuable. PL/I provides such a system. Actually PL/I's preprocessor language is not identical to its execution-phase language, there being a few additions and a few omissions, but the languages are very similar.

A programmer is seldom given very much control over compile-time computation. A few languages provide compile-time variables (symbolic constants), but the use of these variables is usually restricted to a single assignment from a compile-time expression comprised only of constants and other compile-time variables.

If a language provides full algorithmic control at compile time, then a preprocessor phase becomes largely unnecessary. Compile-time conditional control structures can be used to replace preprocessor conditionals. That is to say, if the conditional expression of an if-then-else construct can be evaluated at compile time, and the inaccessible code eliminated, the effect would be the same as conditional compilation provided by a preprocessor phase. Similarly, a loop whose index is a compile-time variable can be used to unravel a loop at compile time. Compile-time input and output should also be provided so that a user can interactively control the compilation options or be informed of the progress of the compilation.

Any execution-phase procedure provided by the programmer can be executed at compile time provided that that procedure uses only local symbols or compile-time global symbols, and does no execution-phase I/O. In this fundamental way the compile phase differs from the preprocessor phase, since the execution-phase algorithm is not fully specified until the end of the preprocessor phase.

The benefits of doing calculations at compile time rather than at execution time can be substantial. All calculations not involving execution-phase I/O can be performed once at compilation time, thereby reducing the space and time required by the execution phase. This reduction would chiefly benefit production programs that are to be run repeatedly. Compile time I/O can be used to customize a program on each compilation, or report version information during compilation.

The link and load phases have been largely ignored by programming language designers, except for some assemblers, and perhaps rightly so. Although it would be beneficial in some critical system applications for a programmer to have control over computations done in the link or load phases, in most situation this would add needless complexity and operating-system dependence to a language. It would therefore be preferable to allow the compiler to determine what calculations will be done at link time and at load time.

The trend has been toward greater independence in the program-processing time scale. Features that once were available only on one phase are being spread to other phases. An example of such a feature is memory allocation, which in early languages such as FORTRAN was done exclusively at compile time, but is now commonly available at execution time also. Another example is type checking, which is being investigated for the possibility of performing it in any phase (*flexible type checking* is discussed by Heering and Klint.[29]) No type of computation should be considered inherently specific to a particular processing phase.

The conclusion of this section is that if a programming language provides independence of the compile phase and execution phase, then the needs of the elements of this classification will be reasonably well met.

4.b. The Project-Development Time Scale

A programming project advances through several stages of development:

1) design
2) coding
3) prove correctness
4) program entry
5) testing and debugging
6) production use
7) maintenance and enhancement

Supplying all of these phases with the same computational power is easy. All one need do is ensure that the same programming language is used for all the phases. This is almost always the case since very few programming projects change languages in mid-course. Meeting the special needs of all of these phases is more difficult. Let us look at each phase individually.

The special needs of the **design phase** depend on the design methodology used, and in some design methodologies the target language is not considered. It seems, however, that a hierarchical language—one in which low-level processes can be grouped into a higher-level construct—is universally valuable. This property is usually provided by procedures.

During the **coding phase**, one wishes to minimize programming effort. Halstead[26] and later workers[30] have developed metrics for measuring the complexity of programs and have used these metrics to compare the *language level* of various programming languages. Language level is a measure of the complexity of code needed to express an algorithm using particular language, and is inversely proportional to programming effort in that language.

There are certain language constructs that inhibit **proof of correctness**, and others that aid it. (Methods for proving correctness are summarized by Elspas et al.[16]) But, since proving correctness is still not a popular activity, one could simply say that if one wishes to prove correctness, one should make use of only the subset of the language that is amenable to such proofs.

It should be mentioned at this point that a proof-of-correctness phase will not always eliminate the debugging phase. Many bugs arise from human error in the use of a programming language, such as incorrect application of operator precedence rules, mistaken use of operators (e.g. using an assignment instead of an equality test in C), or missing statement terminators (semicolons in Pascal). These misconceptions will persist through the proof of correctness, hence the programmer will prove his concept correct, not his program. There are also theoretical limits on proofs of correctness.

The **program entry phase** benefits, as does the coding phase, from a concise language. Nevertheless, a language that is so concise that it is cryptic can impede program entry. If the language makes extensive use of characters that are distinguishable only by subtle differences in typography such as "`1`", "`l`", "`I`", "`|`", and "`!`", then it can be difficult for anyone except the original coder to type the program correctly.

In the **testing and debugging phase** it is desirable to have a great deal of feedback from the program during execution. This feedback can be obtained by using an interactive interpreter. When an interactive interpreter is not available, a programmer must resort to a symbolic debugger or hand coded debugging statements. An interpreter that incorporates a source editor, like those common in BASIC and APL interpreters, has the advantage that it can provide a shorter revision cycle (test, diagnose, correct, retest) than a symbolic debugger.

Many interpreters require that if any part of a program is being interpreted, then all parts must be interpreted. It would sometimes be advantageous, however, to invoke compiled and tested procedures from libraries during the interpretive debugging of the calling module. Similarly, it is sometimes desirable to debug a procedure interpretively, which was invoked by a compiled module. As a result, an interpreter should be able to dynamically load and execute compiled procedures, and a compiler should be able to invoke the interpreter to run source code.

The **debugging phase**, the **production-use phase**, and the **maintenance phase** are the ones that benefit from error resistance in a programming language. Gannon and Horning,[18] and Ripley and Druseikis[46] have analyzed the error resistance of some language constructs, and present a survey of papers on error resistance. One of the goals of structured programming techniques[12, 59] is to increase the error resistance of programs.

For the **production-use phase** one would desire a language that produces compact and efficient machine code. The cost of hardware goes up significantly with its speed and memory size, so a small fast program means reduced costs. Furthermore, in the commercial-programming world, the size and speed of a program product can directly affect its market share. Speed during production use is where a compiler is superior to an interpreter.

Not all of these phases have the same importance, and in different programming shops the relative importances vary. In research environments, the production-use phase is often non-existent and emphasis is placed on the design, coding and debugging phases, which may be mingled rather than distinct. Thus, a research shop would prefer a language with a good interactive development environment such as LISP[47] or APL. In a commercial programming shop, the performance of their end product in the production-use and maintenance phases are the most critical. Hence, such shops would tend to use a language such as FORTRAN, COBOL, or C, that produces efficient, maintainable programs. When the production use phase outways all the others, a shop may choose to program in assembler. (Sometimes, of course, the choice of languages is based on other factors, or has no rational explanation.)

The conclusion of this section is that the characteristic of a language that contributes most to independence in this time scale is that the language be both interpretable and compilable, that the interpreter be interactive, and that interpreted code should be able to invoke compiled code and vice versa.

4.c. The Language-Evolution Time Scale

Computer science has changed radically in its brief history, and programming languages have changed with it. Since it is a young science, one can expect it to keep changing. If a programming language is to remain applicable in the future then one must allow it to change, but should provide guidelines for change. Two general approaches have been taken in providing guidelines for change: the FORTRAN approach and the ALGOL approach.

The FORTRAN approach has been that when the language is changed all previous programs should still run correctly without modification. This guideline has been fairly well followed right up to and including FORTRAN 77, except for two small changes to the semantics of FORMAT statements and DO loops.*

The ALGOL approach has been that when an ALGOL-like language is changed it is given a new name, and all programs in the old language are discarded or manually translated to the new language.

Clearly, neither of these two approaches is satisfactory. The FORTRAN approach is too restrictive and leads to a patched up language from which obsolete features are never removed. The ALGOL approach can also be rejected as too costly. A massive investment in software cannot be easily discarded.

The most common manifestation of the problems caused by time-dependent programming languages is when a programming shop receives a new, enhanced compiler from a manufacturer only to find that it will not compile programs written for the old compiler. The problem is so aggravating that Grace Murray Hopper[33] has proposed the death penalty for language implementors who fail to supply program updaters with new compilers. Perhaps this penalty is too extreme.

The time independence of FORTRAN and some of the design freedom of ALGOL-like languages could be achieved by allowing extensive modifications to the language but requiring that all existing programs can be *machine* translated into the new language version. If the old programs cannot be fully machine translated, then obsolete language features should be at least machine recognizable as such. If neither of these conditions can be met then the language should be given a new name. Basically, no language enhancement project would be complete until an automatic program updater was also written.

* This approach is expected to be abandoned in the next FORTRAN standard.[3]

By using translators, *backward* time independence could also be achieved. It could be made a requirement that all language enhancements be accompanied by an automatic translator for converting programs in the enhanced language back into the older dialect. It should be possible to write such translators, provided that the old language was a suitable target language for automatic translation, although efficiency and size of the program may suffer. With such translators available, programs developed in the enhanced language could go into wide circulation even before compilers for it had been developed on all machines. Furthermore, obsolete machines for which system software development had ceased could still benefit from new software.

The subsection on implementor independence (3.d.) above, contained a discussion of the benefits of allowing implementors to change a language. It should be noted, however, that the translators proposed in that section to maintain compatibility between implementors are different from the ones proposed here to maintain compatibility between language versions. The relationship of these two types of translators would resemble the branches of a fir tree. The implementor-compensating translators would translate different implementor versions to or from a central trunk version, and the language-evolution-compensating translators would translate programs up and down the trunk of the tree. The importance of controlling both the width and height of the translator tree is discussed later, in the section "Managing the proposed language-version translators."

5. Planes of Independence

Very few language features affect independence in only one dimension; most have implications in two or more dimensions at the same time. An analysis of independence that considers two dimensions at once would be treating independence in a two dimensional plane. From the four dimensions of independence presented here one can obtain six such planes of independence. If, however, the classifications presented earlier are applied to each dimension before forming the planes, then the dimensions can be combined in 84 different ways. Since there are so many of them, and since their interactions are often very subtle, it would not be reasonable to discuss all 84 classified planes here.

There is however one plane of independence that is treated extensively in existing literature, though not called by that name. That plane is the man-machine plane.

5.1. The Man-Machine Plane

In the literature, the man-machine plane is usually reduced more or less to a single line. This is done by reducing the human dimension to a single point representing an average human and the machine dimension to a single point representing the average digital computer, and then considering independence in the line joining these two points. (If standard deviations for the group of humans are also presented, then in a sense a triangular region of the plane is being treated rather than just a straight line.) For a survey of the psychology of human-computer interaction see ACM Computing Surveys,[43] Vol. 13, No. 1.

5.1.1. or 1.e. Digital and Human Computers

In a sense humans should have been included in the discussion of machines (the first dimension) because programs are run, not only by digital computers, but also by human beings. Often a programmer will run parts of a program in his head, or with a pencil and paper, many times before a machine ever runs it. Therefore, when one designs a programming language, one should design it not only for execution by machines, but also for execution by humans.

There are some significant differences between the ways that a human executes an algorithm, and the way a machine does. Humans often apply induction to predict how a calculation will proceed, whereas computers simply carry out the entire calculation just as it is described. As a result, programming-language constructs that help a programmer visualize the inductive step in a program, such as **FOR** loops, assist a programmer in his simulation of its execution. They also assist inductive proofs of correctness.

Another difference between human and digital computers is that humans executing a program would rather apply a high-level operator conceptually than apply the series of low-level operators that comprise it. Computers would "prefer" a long sequence of low-level operators (one can at least say that computers that apply low-level operators are easier to build than ones that can apply high-level operators.) It is the preference for high-level operators by humans that lead to the development of high-level programming languages.

When a human is the "target machine" of a program it is often not for the purpose of running the program, but rather to *understand* the program. Algol 60 was the first language designed with the stated goal of being suitable for the publication of algorithms for humans.[44] The reason for the development of structured programming techniques,[12, 59] and of languages that support such techniques, was to increase human understanding of programs.

The WEB system[37] was designed by Donald Knuth with human-understanding of programs as one of its principal objectives. When a human is to make use of an algorithm, he would like to know not only *how* to perform each step of a computation, but also *why* each step is done. It is the purpose of comments to supply this information. Almost all programming languages permit comments, but since they do not exercise any control over the contents of the comments, comments themselves cannot be considered part of the language. In the WEB system, however, the comments are forced to reflect the structure of the program being described, and hence are formally part of the programming language.

5.1.2. Implementors and Machines

There is a great deal of similarity between the analysis of independence in this plane and the analysis of independence in the human dimension under the classification *users and implementors*. This is because the needs of the implementor are very similar to the needs of the machine. The reader should therefore consult that section for a longer discussion of the topic. There are, however, some differences, in that some things that are easy to implement take too long to execute. The original version of LISP is an example of a language that is implementor oriented, but too slow to be considered machine oriented.

6. Common Requirements of the Dimensions

Although independence in the four dimensions sometimes places competing demands on a language designer and implementor, there are also properties of a language that would contribute to independence in more than one dimension at once. One of these properties of a language is that it be both compilable and interpretable. This property would benefit independence of machine size (1a), and system mode (2c), and independence in the program processing (4a) and project development (4b) time scales. Almost any language can be either compiled or interpreted, but the preference is for a language that is *efficiently* interpretable in an interactive environment and *efficiently* compilable. The language should be efficiently executable from source text, as is a command language, and should not require multiple passes over the source code. At the same time the language should be compilable into executable code that does not require extensive run-time variability.

Another property that contributes to independence in more than one dimension is flexibility of procedure invocation. This property would benefit independence of discipline (2a), system mode (2c), problem-solving methods (2d), user qualifications (3a) and the program-processing time scale (4a). Procedure invocation should permit both positional and keyword parameters, and variable length and variable form parameter lists. Operator overloading would be desirable as would the ability to define new operators (extensible syntax).

To minimize the size of the various translators proposed, the language should have a small and orthogonal grammar. Procedure invocation should be used as much as possible, rather than distinct constructs.

7. Managing the Proposed Language-Version Translators

We have repeatedly proposed the use of program translators to achieve independence. They have been proposed for four classifications in three different dimensions: independence of machine size (1b), language evolution (4d), implementors (3d), and natural languages (3b). If this technique is applied indiscriminately, it could result in a labyrinth of translators to get from one dialect to another. Therefore, controls should be placed on the number of language versions generated, and a new version should be created only with good reason.

One of the translators proposed was for providing implementor independence. The implementor of a language dialect should be responsible for writing the translators to and from the standard language. This responsibility would provide him with incentive for making his dialect very similar to the standard language, or to implement the standard language unchanged. Each of the two translators should be written in its destination dialect, that is, the translator from dialect A (the standard language) to dialect B should be written in dialect B and vice-versa. Then, translators in the starting dialect can be obtained for free by running the translators in the destination dialects through each other. If one dialect is a proper subset of the other, then only one translator would be needed.

The language-standards committee should supply the translators that provide machine-size independence. English should be chosen as the natural-language base for a programming-language standard, and separate language standards committees should be established to provide a dialect for other natural languages. Those committees should supply the translators for converting those dialects to and from the English standard.

Every program intended for sale or publication should start with a header that identifies the architecture, implementor, machine size, peripheral devices, operating system, natural-language group, and language-evolution version that the program is intended for. (Such a header would be valuable even for existing languages.) Then if a program were submitted to the wrong compiler, the programmer would be notified, and he could select the appropriate sequence of translators needed to convert it into the correct version. It seems that it would even be possible to write an automatic version-translation utility which accepts a program along with its current version designator, accepts a designator of the desired version, selects the correct sequence of translators needed to perform the transformation, and applies that sequence.

Since the same program may be valid, unchanged, for many different compilers, a program should be able to list the version codes of all of the compilers that will accept it. Using such a list would reduce the number of translators invoked that make no change to a program.

This discussion seems to suggest such a complex maze of translators that a user could easily get lost, but this is not necessarily the case. Most users do not habitually change environments and hence translations from one language version to another would be done far less frequently than compilation or execution.

8. Conclusion

This paper has presented four dimensions of independence that should be considered when designing or evaluating a programming language. A precise definition of independence has also been presented. In addition it has shown the way in which many language features and language evaluation criteria affect independence in these dimensions.

Is it appropriate to treat independence as a functional in a four space? The mathematical definition of a four space is sufficiently broad to include the one represented by these four dimensions. It allows the dimensions to be discrete or continuous, and does not demand orthogonality of the dimensions. The point being made, however, is that the four dimensions presented here are sufficiently orthogonal that it is reasonable to treat them separately. Since no precise mathematical definition has been given for language independence or applicability, and probably none can be devised for the general case, the analogy with a four space is incomplete.

Is independence in these four dimensions necessary and sufficient for the design of a good language? Independence is certainly not necessary, since many successful languages have been highly dependent in one or more of these dimensions, but we believe that a high level of independence in the four dimensions would be sufficient for the design of a good and useful language.

Are the four dimensions necessary and sufficient for a study of independence? With regard to the necessity of the four dimensions, this paper has referenced considerable existing literature that treats three of the four dimensions (machine, problem, and human), and thus shows that they are widely recognized as important. The fourth dimension—time—is less-well supported by existing literature, but we hope that we have presented sufficient arguments to prove that failing to consider this dimension in designing or evaluating language independence would be a deficiency.

With regard to the sufficiency of the four dimensions, this paper has not held that independence in the four dimensions would be sufficient for a study of independence. It may well be that more dimensions or more classifications of the existing dimensions are needed. We believe, however, that the four dimensions presented here, and the accompanying classifications of these dimensions, treat the majority of independence issues.

9. Acknowledgments

The author would like to thank Gordon Cormack, Thomas Strothotte, Michel Devine, Doug Dyment, Benton Leong, Spencer Murray, Don Cowan, Dave Boswell, Bruce Simpson and especially Gregory J. E. Rawlins for their criticisms, suggestions, and encouragement in the preparation of this paper. The author would also like to thank Wendy Goodwin for her valuable editorial suggestions.

10. References

1. United States Department of Defense. *Reference Manual for the Ada Programming Language*. (1980).
2. *MACSYMA Reference Manual*. The Mathlab Group, Laboratory for Computer Science, MIT (Jan. 1983). Two volumes
3. ANS X3J3, "Proposals accepted for future Fortran." Standing Document S6.86. (May 1983).
4. Abelson, Harold. and di Sessa, Andrea A., *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, Cambridge, Mass. (1981).
5. Aho, Alfred V. and Ullman, Jeffrey D., *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts (1978).
6. Backus, John., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *Communications of the ACM*, Vol. 21, No. 8, pages 613-641 (Aug. 1978).
7. Brown, Peter J., *Macro Processors and Techniques for Portable Software*. John Wiley & Sons, London (1976).
8. Brown, Peter J., *Software Portability: An Advanced Course*. Cambridge University Press, Cambridge, England (1979).
9. Char, Bruce W., Geddes, Keith O., Gentleman, W. Morven., and Gonnet, Gaston H., "The design of Maple: a compact, portable, and powerful computer algebra system." in *Proceedings of the 1983 European Computer Algebra Conference* (1983).
10. Clocksin, William F. and Mellish, Christopher S., *Programming in Prolog*. Springer-Verlag, Berlin (1981).
11. Cormack, Gordon V., "Extensions to static scoping." *ACM SIGPLAN Notices*, Vol. 18, No. 6, pages 187-191 (June 1983).

12. Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., *Structured Programming*. Academic Press, London (1972).

13. De Remer, F. and Kron, H., "Programming-in-the-large versus programming-in-the-small." *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, pages 80-86 (June 1976).

14. Donahue, James. and Demers, Alan., "Data types are values." *ACM TOPLAS*, Vol. 7, No. 3, pages 426-445 (July 1985).

15. Eastman, C. M., "A comment on English neologisms and programming language keywords." *Communications of the ACM*, Vol. 25, No. 12, pages 938-940 (Dec. 1982).

16. Elspas, Bernard., Levitt, Karl N., Waldinger, Richard J., and Waksman, Abraham., "An assessment of techniques for proving program correctness." *ACM Computing Surveys*, Vol. 4, No. 2, pages 97-145 (June 1972).

17. Ganapathi, Mahadevan., Fischer, Charles N., and Hennessy, John L., "Retargetable compiler code generation." *ACM Computing Surveys*, Vol. 14, No. 4, pages 573-592 (Dec. 1982).

18. Gannon, John D. and Horning, J. J., "Language design for programming reliability." *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, pages 179-191 (June 1975).

19. Garey, Michael R. and Johnson, David S., *Computers and Intractability, A Guide to the Theory of NP-Completeness*. p. 65, W. H. Freeman and Company, New York (1979). Section 3.2.1(7) Multiprocessor Scheduling.

20. Gehani, Narain., "Units of measure as a data attribute." *Computer Languages*, Vol. 2, No. 3, pages 93-111, Pergamon Press, Great Britain (1977).

21. Ghezzi, Carlo. and Jazayeri, Mehdi., *Programming Language Concepts*. John Wiley & Sons, Inc., New York (1982).

22. Gilman, Leonard. and Rose, Allen J., *APL: An Interactive Approach. Second Edition*. John Wiley & Sons Inc. (1974).

23. Goldberg, A. and Robson, D., *SMALLTALK-80*. Addison-Wesley, Reading, Mass. (1983-1984). Four volumes

24. Goodyear, Peter., *LOGO: A Guide to Learning Through Programming*. Ellis Horwood Limited, Chichester, England (1984).

25. Hall, Andrew D., "The Altran system for rational function manipulation—a survey." *Communications of the ACM*, Vol. 14, No. 8, pages 517-521 (Aug. 1971).

26. Halstead, Maurice H., *Elements of Software Science*. Elsevier, New York (1977).

27. Harland, David M., *Polymorphic Programming Languages*. Ellis Horwood Ltd., Chichester, England (1984).

28. Harrison, Michael A., *Introduction to Formal Language Theory*. Addison-Wesley, Reading, Massachusetts (1978).

29. Heering, Jan. and Klint, Paul., "Towards monolingual programming environments." *ACM TOPLAS*, Vol. 7, No. 2, pages 183-213 (Apr. 1985).

30. Highland, Harold Joseph, Ed., "The workshop on software metrics SCORE 82." *ACM SIGMETRICS Performance Evaluation Review*, Vol. 11, No. 2 & 3, pages 31-126 & 32-128 (1982).

31. Hoare, C. A. R., "An axiomatic basis of computer programming." *Communications of the ACM*, Vol. 12, No. 10, pages 576-580 (Oct. 1969).

32. Hoare, Charles Anthony Richard., "The emperor's old clothes." *Communications of the ACM*, Vol. 24, No. 2, pages 75-83 (Feb. 1981).

33. Hopper, Grace Murray., "Keynote address, ACM SIGPLAN history of programming languages conference (June 1978)." in *History of Programming Languages*, ed. Richard C. Wexelblat, pp. 7-24, Academic Press (1981). (Exact reference: page 20 paragraph 2.)

34. House, R. T., "A proposal for an extended form of type checking." *The Computer Journal*, Vol. 26, No. 4, pages 366-374, Wiley Heyden Ltd. (1983).

35. Jensen, Kathleen., Wirth, Niklaus., Mickel, Andrew B., and Miner, James F., *Pascal User Manual and Report. Third Edition.* Springer-Verlag, New York (1985).

36. Jones, Simon L. Peyton., "Parsing distfix operators." *Communications of the ACM*, Vol. 29, No. 2, pages 118-122 (Feb. 1986).

37. Knuth, Donald E., "Literate programming." *The Computer Journal*, Vol. 27, No. 2, pages 97-111 (May 1984).

38. Leinbaugh, Dennis W., "Indenting for the compiler." *ACM SIGPLAN Notices*, Vol. 15, No. 5, pages 41-48 (May 1980).

39. Levine, John., "Why a LISP-based command language?" *ACM SIGPLAN Notices*, Vol. 15, No. 5, pages 49-53 (May 1980).

40. Männer, R., "Strong typing and physical units." *ACM SIGPLAN Notices*, Vol. 21, No. 3, pages 11-20 (Mar. 1986).

41. Marcotty, Michael., Ledgard, Henry F., and Bochmann, Gregor V., "A sampler of formal definitions." *ACM Computing Surveys*, Vol. 8, No. 2, pages 191-276 (June 1976).

42. McCarthy, John., "History of LISP." in *History of Programming Languages*, ed. Richard C. Wexelblat, pp. 173-185, Academic Press (1981). Presented at the ACM SIGPLAN History of Programming Languages Conference (June 1978).

43. Moran, Thomas P., Guest ed., "Special issue: the psychology of human computer interaction." *ACM Computing Surveys*, Vol. 13, No. 1, pages 1-141 (Mar. 1981).

44. Naur, Peter, Ed., "Revised report on the algorithmic language ALGOL 60." *Communications of the ACM*, Vol. 6, No. 1, pages 1-17 (Jan. 1963).

45. Organick, Elliot I., *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, Mass. (1972).

46. Ripley, G. David. and Druseikis, Frederick C., "A statistical analysis of syntax errors." *Computer Languages*, Vol. 3, No. 4, pages 227-240, Pergamon Press, Great Britain (1978).

47. Sadewall, Erik., "Programming in an interactive environment: the "LISP" experience." *ACM Computing Surveys*, Vol. 10, No. 1, pages 35-71 (Mar. 1978).

48. Teitelbaum, Tim. and Reps, Thomas., "The Cornell program synthesizer: a syntax-directed programming environment." *Communications of the ACM*, Vol. 24, No. 9, pages 563-573 (Sept. 1981).

49. Teitelman, W., *INTERLISP Reference Manual.* XEROX Co., Palo Alto, Calif. (1978).

50. Tennent, R. D., "The denotational semantics of programming languages." *Communications of the ACM*, Vol. 19, No. 8, pages 437-453 (Aug. 1976).

51. Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., and Fisker, R. G., *Revised Report on the Algorithmic Language Algol 68.* Springer-Verlag, Berlin (1976).

52. Wadge, W. W. and Ashcroft, E. A., "LUCID: a nonprocedural language with iteration." *Communications of the ACM*, Vol. 20, No. 7, pages 519-526 (July 1977).

53. Wadge, William W. and Ashcroft, Edward A., *Lucid, the Dataflow Programming Language.* Academic Press, London (1985).

54. Wallis, Peter J. L., *Portable Programming.* The Macmillan Press Ltd., London (1982).

55. Wegner, P., "The Vienna definition language." *ACM Computing Surveys*, Vol. 4, No. 1, pages 5-63 (Mar. 1972).

56. Wells, Mark B., "A potpourri of notational pet peeves (and their resolution in Modcap)." *ACM SIGPLAN Notices*, Vol. 21, No. 3, pages 21-30 (Mar. 1986).

57. Wolberg, John R., *Conversion of Computer Software*. Prentice-Hall Inc., Englewood Cliffs, New Jersey (1983).

58. Xenakis, John., "PL/I-FORMAC interpreter." in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation (Los Angeles, March, 1971)*, ed. S. R. Petrick, pp. 105-114, ACM, New York (1971).

59. Yourdon, Edward Nash, Ed., *Classics in Software Engineering*. Yourdon Press, New York (1979).