THE DESIGN OF USABLE PROGRAMMING LANGUAGES*

Michael Hammer
Project MAC
Massachusetts Institute of Technology
Cambridge, Massachusetts

A methodology for the design of programming languages is set forth. The principal objective of this approach is the development of languages that are easy to use; it is argued that conventional language designs do not satisfactorily achieve this goal. The basic principles of the proposed approach are restriction and discipline: by appropriately limiting the programmer's freedom of choice, the number of decisions he must make can be reduced. In particular, it is argued that languages should be designed for individual problem areas, and that each language should be built around a style of problem solving, an algorithmic structure appropriate to its application domain. Principles of declarative and data-oriented programming, which avoid a processor-oriented view of computation, are also set forth.

1. INTRODUCTION

By now the celebrated "software problem" is a commonplace. It is generally recognized that the cost and difficulty of constructing and maintaining reliable software is the major impediment in the way of dramatically increased and more effective application of computers [1]. In response to this problem, there has been much activity of late in "software engineering", in the development of programming methodologies and standards, and of languages to support them, in order to ease and streamline the programming process.

However, many of the proposals being set forth seem to be concerned with developing improved techniques for using conventional tools, rather than with designing new and different tools. The primary thrust of most research in "software engineering" has been the development of principles for the use of so-called general purpose, higher level programming languages, the use of which should result in better designed and more reliable software. By setting forth elements of style that a programmer should follow in utilizing the powerful and basically unstructured facilities of a general purpose language, the hope has been for the developement and use of some limited standard techniques for program construction. Structured programming [2], modular and top-down design [3,4,5], and goto-less programming, for example, are all methodologies intended to encourage a structured and disciplined use of programming languages, in contrast to indiscriminate coding. The development of such protocols has spawned research in the design of the precise set of basic linguistic features that support and encourage the chosen style of programming [6,7,8,9].

These emerging ideas and theories certainly make an important contribution to the evolution of a science of programming languages. But much of the current research is built on certain tacit assumptions about the nature of programming; these unspoken attitudes have major consequences for the way languages are designed and evaluated, and deserve examination.

2. GENERALITY AND FLEXIBILITY

Primary among these postulates is a strong emphasis on <u>generality</u>. The domain of discourse of software engineering is almost always software systems construction in general, rather than in any one limited area. The goal of most work in this area seems to be the development of what might be termed "algorithmics", a science of devising general languages and techniques to describe all kinds of computational algorithms. In this context, any problem is (almost) as important as any other one. As a consequence, a high premium is put on flexibility and power as a criteria for judging programming languages. That is, a language is expected to provide the basic facilities for expressing all kinds of algorithms; it may encourage or support certain very broad styles of programming, but it is not supposed to confine the programmer within these loose constraints. In these lights, a language must be flexible enough to adapt to any use to which the programmer wishes to put it; it must not inhibit or inconvenience him in the selection of a problem or an algorithm to solve it. A language offers the programmer basic computational units, but it may not tell him how they are to be combined into whole programs. Constructing algorithmic specifications from computational primitives is seen to be the province of the programmer, the act of programming, on which the language and its designer are not to intrude. A programming language, in this view, must offer the right kind of support, and provide the best stage, for the exercise of the programmer's talent. This attitude is related to the vision of programming as an "art" [10], a forum for the exercise of inventiveness and creativity. It follows that a language designer should encourage a programming artist to utilize sound programming practices, but must not unduly interfere with the exercise of his craft.

As a consequence, most of the languages that are surfacing in the wake of the software engineering movement, are so-called general purpose, higher-level languages, usually derivative of ALGOL or LISP. The novel features of these languages tend to be rich control structures and definitional mechanisms that allow a programmer to define the data structures he wants to use and the procedures appropriate for their manipulation. While the particular details and syntax of these languages vary greatly from one to another, they are all motivated by the view of programming set forth above, and share a common view of the computation process as well.

Though contemporary programming languages are called "higher-level", the model of computation which they

*This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA order No. 2095 and by ONR Contract No. N00014-75-C-0661.

embody is in reality a very low-level one, and not far removed from that provided by classical von Neumann architecture machines (and their formal mathematical models). In particular, most of these languages have a processor/memory model of program execution. The abstract computing device which defines the semantics of a language is assumed to possess a central processor which performs operations on values, and a memory wherein these values are stored. A program is viewed as a list of processor instructions, which are sequentially interpreted; at any point in time, the status of the computation specified by the program can be described by the locus of control in the program and by the contents of the memory. The processor operations tend to be binary arithmetic and logical operations, similar to those available on most real machines. Most of the "high-level" aspects of such languages tend to be purely syntactic in nature, their semantic capabilities being decidedly "low-level". Even much vaunted, "high-level" control structures such as do-while, case, and recursive procedure calls, are nothing more than syntactic sugar, macros locally expandable in terms of basic machine instructions.

3. PROBLEM ORIENTED LANGUAGES

Another approach to the design of programming languages has been that of domain specific languages each applicable to a relatively narrow range of applications [11,12]. Of course, even "general purpose" languages, while Turing complete, are best suited for some particular problem area; but these areas tend to be very broad, such as "non-numeric computation". The domain of application of a specialized application language is much narrower.

Most domain-specific programming languages can be categorized in one of two ways; either as a "sugared"" general purpose language, or as a data presentation language. In the former case, the language designer investigated the chosen problem domain and determined the data types and structures, and operations on them, that are fundamental to the field; these features were then grafted onto some general-purpose language of the designer's choosing. In the result, it is argued, the application programmer has available to him the basic constructs appropriate to the domain, which he can manipulate as desired.

The foregoing scenario has often been used to justify the development of extensible language systems, for the facilities of such systems promote this kind of language development. But this scheme does not really produce a domain specific language. The resulting "specialized" language is just a mildly disguised version of the general purpose language. Nothing fundamental to the problem domain has been deeply embedded in the language.

This is not to say that this scheme is without merit. It does represent the rudiments of a methodology for adapting a general purpose language to use in a narrow domain, in order to produce more reliable programs. The definition and maintenance of a level of abstraction is an important step in reducing the complexity of general purpose programming. But it is not the same as building a language tuned to the problem area. In using an abstraction, the basic character and structure of the supporting general purpose language is not tampered with; the style of programming, in fact the style of problem solving, embedded in and supported by that language remains unchanged. The applications programmer is indeed provided a few specialized units but must build his own structure into which to fit them. It remains the case that, in order to devise and describe his algorithm, the applications programmer must think in terms of abstract processors that are not far removed from Turing machines; the only improvement is that this processor can perform some special operations that the programmer will frequently use. But the organization and structure of a user program is not at all specialized; it is built out of the same general building-blocks that apply to every problem domain.

A few specialized data types and operators do not constitute a specialized language. We believe that the basic ways in which the algorithms in a specialized domain are composed, in which the various individual units are tied together, are of a specialized character and are peculiar to the problem domain; and a domain-specific language should incorporate these algorithmic structures. Furthermore, the algorithmic structure of many an application domain, the problem solving methodology most natural to the field, cannot be easily described in terms of the computational model provided by general purpose languages.

The other classical model for domain-specific languages is that of the so-called "problem-oriented" languages. Here, the language designer so narrowly restricts the domain of application that he is able to exhaustively list all the application-related actions which a user might need. No capabilities for the construction and representation of algorithms are provided in the language; there is no need for them, for the designer has fully anticipated all the user's needs. The user will only be able to specify which actions he wants to perform, in what order, and to what data. These languages boil down to what may be called data presentation languages; in their use, the user does little more than parametrically activate certain pre-canned routines.

It is to this class that most non-general purpose languages belong [13,14]. For example, in the civil engineering language STRUDL [15], a user describes a structural situation, using problem-oriented terms such as GRID, TRUSS, FRAME; he then may ask for the load, stress, strain, etc. to be computed at various points in the structure. Some problem-oriented languages are more sophisticated than this; but to call them programming languages, is to push the term to its extreme. In reality, "programs" in these languages are just formatted data to an application package. The problem domains addressed by these systems are too small to make this approach widely applicable.

4. VERY HIGH LEVEL LANGUAGES

Recently, there has been a commendable trend towards the design of so-called "very high level" or "non-procedural" languages, intended to ease the programming task [16,20]. These languages attempt to reduce the level of detail specified by a programmer by making available constructs that facilitate expression of intent. Some of these features constitute a real departure from the tradition of von Neumann computer architecture. These languages variously provide non-machine-oriented aggregate data structures (such as sets and relations), together with facilities for accessing and manipulating them (such as implicit loops, implicit data structures, and associative referencing) [17,18,19].

While such features do provide for significant local simplification in the expression of algorithms, much of the current effort in very high level language design is directed towards the development of general purpose very high level languages. This research hopes to discover a basic set of non-computer-oriented primitives that will suffice for the expression of all algorithms over a wide range of applications. The overall structure of such languages is usually reminiscent of ALGOL, with the nonprocedural elements added on to a conventional base. Thus, the very high level constructs make it easier to express individual parts of an algorithm, but do not have much impact on the global construction of programs.

5. AN ALTERNATIVE APPROACH

We should like to propose another methodology for programming language design, one that is based on different postulates and assumptions about the programming activity. Our main interest is in designing simple programming languages--meaning languages that are easy to use and that lead to programs that are easy to read and modify.

It is our intent to reduce the ingenuity, cleverness, and creativity necessary to write good programs. We avoid the view of programming as a stimulating intellectual activity; while attractive, this is a grossly uneconomical vision. We want to propose the notion of most programming being very routine and predictable, with only novel and complex new problems requiring the exercise of creativity in the programming itself. While not an exciting attitude, this approach to programming can have enormous impact on the cost of building and maintaining software.

The basic principles which we shall apply to achieve our goals are restriction and discipline. We shall generally be willing to forego the benefits that accrue from flexibility and freedom, and shall impose limitations and structure on programmers and the languages they use. One of the principal sources of difficulty in programming lies in the large number of choices which a programmer has to make, particularly with respect to issues that have no significance in the problem area being addressed. Wherever we can, we shall reduce the number of decisions that the programmer has to make and narrow the range of alternatives from which he must choose, even at the cost of restricting the applicability of the language or limiting the programmer's freedom.

Accordingly, we abandon the notion of a "general purpose" programming language; the first step in reducing the complexity of programming is to reduce the individual areas of application of the languages to be used. A language that is equally convenient for a wide variety of problems is also equally inconvenient for all of them. We accept the inevitability and even the desirability of a variety of languages, each tuned to a particular need.

The domain of application of a programming language should be a problem area that is characterized by a logical and algorithmic consistency. Specifically, there must be a commonality to most of the problems in the area, in that a standard problem-solving methodology will suffice for all of them. That is, though there will be many different algorithms employed to solve many different problems, all these algorithms will have a common structure. Such a problem domain may be large or small; it is characterized not by size but by a conceptual unity. Two example areas might be business data processing and discrete systems simulation.

Having chosen such a problem area, the first step in designing a language for it is to identify the common algorithmic structure that unifies the domain. This problem solving regime will be embedded in and enforced by the language. The language user will not only be provided with the appropriate computational primitives for constructing his algorithm, but he will also be instructed and guided by the language as to an appropriate structure for his algorithm, how these primitives should be put together. Thus, in the initial stage of algorithm formulation, the programmer will not be on his own; rather, the structure of the language will almost force him to organize his program in a specific way. A major area of programmer choice, and hence a source of much difficulty and potential error, will be greatly diminshed in scope.

Given that this algorithmic structure is embedded in the language, it might be very difficult for a programmer to solve a problem using an algorithm that does not fit into this pattern. Furthermore, it will undoubtedly be the case that for some problems the provided structure will be inappropriate. However, so long as the predetermined discipline suffices for most problems in the area, we shall be content. For those problems for which the language is unsuitable, general purpose languages remain available.

We are not moved by the complaint that we are restricting the programmer's degrees of freedom and forcing him to adapt to a structure not of his own choosing. Nor are we concerned about those programmers who see programming primarily as an opportunity for exercising their own ingenuity. We firmly believe that most potential computer users do not fit this description, and that they view computer usage and algorithm specification as an ordeal, not as fun. These people will be grateful for any help they can get, any guide or discipline to their thinking that is provided them. We believe they will much prefer being given the broad outlines of an algorithm which they have to fill in (even if that algorithm is not the one they would have chosen in a vacuum), instead of having to express an entire algorithm themselves, using primitive terminology.

We believe that by restricting the choices that the programmer must make, we significantly contribute to the ease of using a programming language. Such ease of use is not identical to, and may in fact be antithetical to, ease of learning the language. In learning a supposedly simple language such as BASIC, a programmer easily familiarizes himself with the individual features of the language; but to combine them into a whole program accomplishing some useful task is a task of great complexity, the full burden of which rests on the programmer. (The same holds true for PL/I or ALGOL, but there it is difficult even to learn the individual features.) To learn a language such as we are proposing may require significant effort; the user has to learn a whole problem-solving methodology. However, once this is learned, it can be applied again and again, without requiring creativity on the user's part. The complexity of the programming process has been redistributed so that the language carries more of it than is customary. Such a language may be harder to learn, but it should be easier to use in its intended application domain.

An analogy might be made to the design of tools to be used in housing construction. One approach would be to design the best possible set of basic tools and materials, as well as general instructions for their use; with such a set one could build any kind of structure, but to do so one would have to be an accomplished carpenter. An alternative approach would be to design a set of basic prefabricated modular units, and a general schematic plan for a certain kind of house; then it would be straightforward for a less proficient craftsman to construct a building of this type (while almost impossible for anyone to build any other type of structure using the given tools).

In our scheme, "knowledge" of the problem area is contained in the programming language; the knowledge is of good standard algorithmic structures to be used in writing application programs. This approach has in fact been partially instantiated in a number of languages, such as RPG [21], GPSS

[22] and PLANNER [23]. In each case, a preferred model of problem-solving for the domain was selected, and was built into the language.

PLANNER is primarily intended for artificial intelligence applications. The language design is based on the proposition that backtracking is a natural mode of algorithm structure and organization for most problems in the AI area. The basic PLANNER control structure is pattern directed multiprocess backtracking; this algorithmic organization ties language primitives together into programs. This structure is deeply embedded in the language and may be implicitly used by the programmer. Other algorithmic structures, however, must be achieved in roundabout ways. (We note that for our design program to succeed, the problem solving regime embedded in the language must be a good one. For example, it has been argued that automatic backtracking is not a good structure for the AI application area [24]. The language designer must obviously have a great familiarity with programming in the problem domain.)

Similarly, the structure of the RPG language, with its use of indicators to drive the calculation and production of outputs, provides a skeletal framework in which a programmer can work. The popularity of RPG, despite its many shortcomings, testifies to the appropriateness of this computational structure for many data processing applications [25].

6. DECLARATIVE PROGRAMMING

Having argued that the incorporation of an algorithmic organization into a programming language is a prerequisite for the language's facile use, it remains to be decided how this structure is to be expressed, what kinds of linguistic constructs should carry it. To further simplify the programming process, we shall rely on less procedural and more declarative modes of expression; the latter provide descriptions of the result of a computation rather than explicit instruction to a processor. Our fundamental attitude is that a user understands the data with which he is concerned, and the manipulations which are to be performed on this data, and even how these manipulations are to be organized, provided this organization is expressed in terms of the data. People have trouble with converting a data-oriented algorithm into an algorithm description based on a von Neumann machine model.

The control structures of a program provide a skeleton to support the body of the program, the data manipulations. The use of abstract processor-oriented control structures causes an unfortunate mismatch between the data operations which are performed and the program organization which directs their performance. Thus we need ways to express the structure of a program in terms of the data being manipulated, with the "flow of control" being implicitly specified.

One such mechanism for program organization is the idea of data flow programming [26]. There has been a recent trend towards the development of machine architectures based on the notion of data flowing between nodes in a network, where each node represents a computation to be performed on the data entering it, and a node is activated when it receives inputs. The flow of data is usually characterized as a stream of atoms, with the nodes performing simple operations on them. This idea has been developed primarily to provide a better model of computation for purposes of detecting and capitalizing on the parallelism in a system; and data flow models have been used to express the formal semantics of conventional programming languages. However, the idea of data flow also forms an attractive basis for a user-oriented programming language; and it has been used in a language for operating system construction [27].

One can also build domain-specific languages using this concept. In such a language, the elements of data flowing between nodes could be complex data structures or even aggregates; and the nodes could be drawn from a set of powerful manipulations or could represent functional modules written in some other syntax. (The latter approach has been employed in BDL, a language designed for the construction of business data processing systems [28, 29].) Data flow has proven to be very effective for expressing the organization and global "flow of control" in a program. It is especially powerful for describing the operation of complex systems whose different parts may be working asynchronously. The user can implicitly specify that certain parts of his program are to be executed in parallel, or that others are to wait until the occurrence of specified kinds of inputs, without having to describe in detail how this is to be accomplished. Using this scheme, the user can easily describe the relationships and dependencies among program modules in a meaningful way, rather than resort to control oriented descriptions that simulate the desired process in an indirect fashion.

Control structures which organize local computations can often be formulated in terms of the data being manipulated as well. For example, various languages enable the expression of iteration as "for all elements of an aggregate, do the following", rather than by explicitly incrementing an index. More elaborate iterative structures can also be data driven. For example, a common program structure in many data processing applications is to group together all records in a file with the same value of a given field, and then to perform some fixed procedure on each such group. This capability is directly provided the BDL user for parameterization: he need only specify the source file, the relevant field, and the procedure to be repeatedly applied. Thus the programmer can structure his program directly in terms of subsets of the source file, rather than in terms of an abstract processor performing the subsetting. This mode of expression avoids a number of purely language-related complexities inherent in the conventional specification of such a program organization. For example, expressing this program structure in ALGOL or PL/I requires additional temporary storage, several explicit loops, and assorted other machinery, totally unrelated to the real problem.

Declarative modes of expression can be found for other abstract control structures, especially when this is done within the context of a programming discipline. For example, the conditional construct may be broadly construed as having two kinds of uses: either determining whether or not a specific data item is going to be subjected to a manipulation, or choosing the identity of an operation to be performed. In most languages, these two usages are not distinguished and may be freely intermixed. In BDL, a discipline is imposed which insists that the two be applied sequentially: first the arguments of an operation are fully specified; then the nature of the operation is identified; and then the operation is applied to its arguments. To accomplish the first of these tasks, a purely data-driven construct is employed, which selects the desired arguments from available data items. Of course, this effect could be achieved by using a conventional conditional; but the processor oriented conditional has been replaced by a descriptive capability referring only to properties of the data.

7. EFFICIENCY

In the foregoing, we have concentrated on the convenience of the programmer as the dominant criterion in evaluating a programming language. However, the issue of execution-time efficiency needs to be addressed. Computer time will always have an associated cost; and for any improvements in processor speed, new applications that require its capacity inevitably arise.

The problem we must face is that a language that is easier for a programmer to use because it diminishes his decision-making role, also impedes his ability to tune his program for greater efficiency.

Thus, the need for powerful optimizing compilers becomes especially acute for the kinds of languages we are proposing. Therefore, it is fortunate that there are new opportunities for optimizing programs written in these languages, in ways that are far more powerful than the local transformations applicable to conventional general purpose languages. Since our programs will be written in data oriented, high level terms, it will be feasible for a compiler to "understand" what the program is tyring to do, and generate a good implementation of the intent, based on various global criteria. For example, no PL/I compiler can rewrite a Shell sort into a bubble sort; but a compiler for a language with the SORT command could decide to generate a bubble sort because of known properties of the data. Another new opportunity for optimization derives from the fact that these languages are intended to be used in specialized areas. Thus it is possible to build into the compiler "knowledge" about the area: preferred algorithmic styles, useful programming techniques, and all the intuitive knowledge about tradeoffs that application programmers build up.

8. CONCLUSION

To fully develop the approach outlined above, further research and development efforts are required. First, careful study is needed of areas of computer application to determine appropriate domains for specialized languages. Analyses of these problem domains must be made to isolate the basic computational structures that are peculiar to each domain and that need embodiment in its language. Then it is appropriate to look for the right linguistic features to express these concepts; we have indicated that declarative constructs consitute one promising direction. These language designs must not be done in a vacuum. Studies will have to be performed to see how readily real application systems can be coded using these features, and how typical programmers react to them. In this context, it is appropriate to conduct research on the psychology of computer programming: to determine what basic computational primitives people seem to possess, which they are most facile at using, and which they can readily learn. This multifaceted approach should result in the design of truly usable programming languages that make programming a simpler task.

This methodology was consciously applied during the design of the BDL language. Experience with the language to date indicates that in the domain of business data processing, BDL programs are very easy to write, read, and change.

REFERENCES

[1] B. Boehm, Software and its Impact: A Quantitative Assessment, Datamation 19, 5 (1973), 48-59.

[2] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press, London, England, 1972.

[3] B.H. Liskov, A Design Methodology for Reliable Software Systems, Proceedings of the AFIPS 1972 FJCC.

[4] D.L. Parnas, Information Distribution Aspects of Design Mehtodology, Proceedings of the IFIP Congress, August 1971.

[5] H.D. Mills, Top-Down Programming in Large Systems, in Debugging Techniques in Large Systems, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, N.J., 1971, 41-55.

[6] B. H. Liskov and S. Zilles, Programming with Abstract Data Types, SIGPLAN Notices 9,4 (1974), 50-59.

[7] D. Knuth, Structured Programming with go to Statements, ACM Computing Surveys 6,4 (1974), 261-301.

[8] W.A. Wulf, D.B. Russell, and A.N. Haberman, BLISS: A Language for Systems Programming, CACM 14, 12 (1971), 780-790.

[9] N. Wirth, The Programming Language Pascal, Acta Informatica 1, 1 (1971), 35-63.

[10] D. Knuth, Computer Programming as an Art, CACM 17, 12 (1974), 667-673.

[11] J.E. Sammet, An Overview of Programming Languages for Specialized Application Areas, Proceedings of the AFIPS 1972 SJCC.

[12] J.E. Sammet, Programming Languages: History and Fundamentals, Prentice-Hall, Englewood Cliffs, N.J. 1969.

[13] S. Warshall, AMBUSH - A Case History in Language Design, Proceedings of the AFIPS 1972 SJCC.

[14] S.J. Fenves, Problem-Oriented Languages for Man-Machine Communication in Engineering, Proceedings of the I.B.M. Scientific Computing Symposium on Man-Machine Communications, 1966.

[15] D. Roos, ICES System: General Description, M.I.T. Department of Civil Engineering, R-67-49, September 1967.

[16] B.M. Leavenworth and J.E. Sammet, Overview of Nonprocedural Languages, SIGPLAN Notices, 9,4 (1974), 1-12.

[17] J. Earley, High Level Operations in Automatic Programming, SIGPLAN Notices, 9,4 (1974) 34-42.

[18] J.T. Schwartz, On Programming: An Interim Report on the SETL Project--Installment I; Generalities, Computer Science Dept., Courant Institute of Mathematical Sciences, New York University (1973).

[19] J.B. Morris, and M.B. Wells, The Specification of Program Flow in MADCAP VI, Proceedings ACM 25th Annual Conference (1972).

[20] A. Tucker, Very High Level Language Design: A Viewpoint, Computer Languages 1, 1 (1975) 3-16.

[21] I.B.M. Corporation, System/3 RPG II Reference Manual, SC21-7500-4.

[22] G. Gordon, A General Purpose Systems Simulator, I.B.M. Systems Journal 1, 1 (1962), 18-32

[23] C. Hewitt, Procedural Embedding of Knowledge in PLANNER, Proceedings of IJCAI, London, September 1971.

[24] G.J. Sussman and D.V. McDermott, From PLANNER To CONNIVER: A Genetic Approach, Proceedings of the AFIPS 1972 FJCC.

[25] The Software Battle, System/3 World 1,2 (1973) 8-10.

[26] J. Dennis, A First Version of a Data Flow Procedure Language, Proceedings of the Programming Symposium, Paris, April 1974.

[27] P. Kosinksi, A Data Flow Language for Operating Systems Programming, SIGPLAN Notices 8, 9 (1973), 89-93.

[28] M. Hammer, W.G. Howe and I. Wladawsky, An Interactive Business Definition System, SIGPLAN Notices 9, 4 (1974) 25-33.

[29] M. Hammer, W.G. Howe, V.J. Kruskal, and I. Wladawsky, The Business Definition Language, I.B.M. Research Report, March 1975.