An overview of programming languages for specialized application areas

by J. E. SAMMET
IBM Corporation
Cambridge, Massachusetts

INTRODUCTION

There are more than 165 different programming languages in use in the United States today, where only higher level languages are considered as programming languages. If assembly languages were considered in this total it would obviously be much higher. The total number would be still greater if programming languages in use outside the United States were included. (They are excluded here only because of the difficulty of obtaining accurate and sufficiently detailed information.) As individuals, and as an industry, we should ask ourselves, "What is the reason for this enormous proliferation?", particularly since many of these languages claim to be "general purpose." Some languages do serve a wide variety of users and applications, whereas others are restricted in intended usage. The languages which have few users are usually in that category because (a) they are basically for a narrow application area which has relatively few users* or (b) information about the language has not been widely disseminated, or (c) the language and/or its implementation is ineffective and/or has inadequate support, or (d) the language is implemented only on a computer not widely used. The purpose of this paper is to provide some of the background and perspective on the existence, classifications, and general characteristics of those languages which are oriented toward a specialized application area.

The earliest of the major "specialized languages" seems to be APT, developed at MIT by 1955, for numerical machine tool control. A small program—shown in Figure 1—illustrates in an intuitive way the type of language with which this paper is concerned. It is perhaps unfortunate—but is certainly quite true—

*Some languages which have a narrow area of intended usage may actually have a large number of users, e.g., COGO.

that one of the major reasons for the proliferation of programming languages is that designing and implementing languages are fun, and there is a very large NIH (Not Invented Here) factor that makes even minor deficiencies in an existing language a justifiable cause for the development of a new one. This represents the less productive aspect of the proliferation. However, the important cases (meaning the languages widely used) fulfill a bona fide need for a language that can be used by people who don't really understand programming. The specialized languages help these people work in their own professional jargon. The motivation for the development usually comes when individuals find that for each existing language there are facilities that they want, and which they think the language should legitimately contain, but which are not basically available in the language. It is important to emphasize the "not basically available" aspect, as well as recognizing that there is a value judgment involved on this issue. There are certainly cases where FORTRAN has been used to write payroll programs but it is unlikely that anybody would seriously contend that such usage was appropriate; alternatively, people should not condemn FORTRAN for being ill-suited for writing data processing applications since that was not its intent. Similarly, COBOL has been used to generate differential equation programs, but that is certainly a perversion of its major intent. Thus, in considering whether an existing language should be used for a particular problem, its avowed intent must be kept well in mind. This applies not only to the syntax and semantics of the language itself, but also to the type of the machine or environment for which it was designed. A language suitable for use in a batch system is not necessarily well-suited for use in an interactive mode, even though a compiler or an interpreter for the language can indeed be put into a time sharing system. Similarly, a language which provides very good inter-

From the collection of the Computer History Museum (www.computerhistory.org)
Part Program

**Explanation**

**CUTTER/I**
Use a one inch diameter cutter.

**TOLER/.005**
Tolerance of cut is .005 inch.

**FEDRAT/80**
Use feedrate of 80 inches per minute.

**HEAD/1**
Use head number 1.

**MODE/1**
Operate tool in mode number 1.

**SPINDL/2400**
Turn on spindle. Set at 2400 rpm.

**COOLNT/FLOOD**
Turn on coolant. Use flood setting.

**PT1 = POINT/4, 5**
Define a reference point, PT1, as the point with coordinates (4, 5).

**FROM/(SETPT = POINT/1, 1)**
Start the tool from the point called SETPT, which is defined as the point with coordinates (1, 1).

**INDIRP/(TIP = POINT/1, 3)**
Aim the tool in the direction of the point called TIP, which is defined as the point with coordinates (1, 3).

**BASE = LINE/TIP, AT ANGL, 0**
Defined the line called BASE as the line through the point TIP which makes an angle of 0 degrees with the horizontal.

**GOTO/BASE**
Go to the line BASE.

**TL RGT, GORGT/BASE**
With the tool on the right, go right along the line BASE.

**GOFWD/(ELLIPS/ CENTER, PT1, 3, 2, 0)**
Go forward along the ellipse with center at PT1, semi-major axis = 3, semi-minor axis = 2, and major axis making an angle of 0 degrees with the horizontal.

**GOLFT/(LINE/2, 4, 1, 3,), PAST, BASE**
Go left along the line joining the points (2, 4) and (1, 3) past the line BASE.

**GOTO/SETPT**
Go to the point SETPT in a straight line.

**COOLNT/OFF**
Turn off coolant flow.

**SPINDL/OFF**
Turn off spindle.

**END**
This is the end of the machine control unit operation,

**FINI**
and the finish of the part program.


Figure 1—APT (machine tool control)
active facilities does not necessarily provide the broad flexibility and control normally found (or needed) in batch programs of great complexity.

In order to more explicitly establish the level of language that is being discussed, the defining characteristics of a higher level language are considered to be the following: (1) machine code knowledge is unnecessary, i.e., the user does not have to know the machine instructions available on any computer; (2) there must be good potential for converting a program written in a higher level language onto another machine; that is equivalent to saying that the language must be basically machine-independent (but since we know that no languages to date are completely machine-independent, what is being stipulated is a good potential for conversion to another computer); (3) there must be an instruction explosion, i.e., for most of the statements that the user writes in the higher level language, the computer (through a compilation or interpretive process) should generate many more machine instructions; (4) the notation of the higher level language should be more problem-oriented than that of an assembly language, i.e., it should be more “natural” for the class of problems being solved.

MEANING OF, AND STATISTICS ON, APPLICATION-ORIENTED LANGUAGES

Terminology

Many people refer to special-application-oriented languages as “special purpose” languages. This is misleading since the term “special purpose” really applies to a single or a limited set of objectives. For example, one might design a language that was intended to be very easy to use (e.g., BASIC). On the other hand, a major objective of the language might be easy and rapid compilation (e.g., MAD). Alternatively, the objective might be a language in which it was easy to generate efficient object code. Finally, as a particular type of “special purpose,” a language might be designed to be useful for a specific application area.

Another term which is frequently used for the class of languages discussed in this paper is “problem-oriented.” This is bad terminology, because all programming languages are problem-oriented. The main distinctions pertain to the width or narrowness of the problem area involved.

In the most fundamental sense, all programming languages are “application-oriented.” Every programming language which has been developed has been aimed at a particular class of applications, which may be narrow or broad. In the latter case, the so-called general purpose languages such as PL/I and ALGOL 68 attempt to be suitable for every application area and they clearly don’t succeed. This is not surprising when one considers the enormous variety of uses to which computers are put today, including calculation of space trajectories, medical diagnosis, inventory control, numerical machine tool control, graphic displays, and movie animation. With the sole exception of medical diagnosis, all of these applications have fairly general (e.g., FORTRAN, COBOL) or very specialized (e.g., APT) languages used for their solutions. Hence, it is inaccurate to refer to some programming languages as “application-oriented” while others are not. All programming languages are application-oriented, and the only question that can meaningfully be asked is “what type of application?” The answer can be narrow or broad, and also can be classified by either technique or application area, or both. For example, matrix manipulation is both a technique and a class of applications, depending on your viewpoint. (This is of course analogous to the concept that one person’s program is somebody else’s subroutine.)

In contrast with the technique and/or application of mathematics, one might consider the application area of technical computation with engineering as a major subcategory; underneath the latter can be logical design and civil engineering (which in turn has subdivisions of structural engineering and coordinate geometry). In parallel with engineering we might have astronomy. The area of computations in medicine could be considered a subset of technical computations, or might be considered a major category. It is really up to the individual as to how fine a line he wants to draw.

Table I represents one possible schematic, and the reader can easily develop similar tables to reflect his own approach to problem areas. It is important to note

<table>
<thead>
<tr>
<th>Mathematics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td></td>
</tr>
<tr>
<td>Matrices</td>
<td></td>
</tr>
<tr>
<td>Partial Differential Equations</td>
<td></td>
</tr>
<tr>
<td>Non-numeric</td>
<td></td>
</tr>
<tr>
<td>Matrices</td>
<td></td>
</tr>
<tr>
<td>Partial Differential Equations</td>
<td>Polynomials</td>
</tr>
<tr>
<td>Technical</td>
<td></td>
</tr>
<tr>
<td>Engineering</td>
<td></td>
</tr>
<tr>
<td>Logical Design</td>
<td></td>
</tr>
<tr>
<td>Civil Engineering</td>
<td>Structural Engineering</td>
</tr>
<tr>
<td>Astronomy</td>
<td></td>
</tr>
<tr>
<td>Medical</td>
<td></td>
</tr>
</tbody>
</table>
that the distance of the observer from an application area affects his view of it. Thus, civil engineering is one application area from the viewpoint of a computer scientist, but has many levels and subdivisions for a person involved in building bridges.

Another method of classifying the application areas is through the realization that some types of applications involve a particular specialized discipline or specific technology, whereas others are specialized but can be used across many areas. Illustrations of the former include civil engineering, movie animation, social science, etc., whereas the latter include simulation, graphics, etc. A more complete list is given in Table II.

This paper specifically deals with, and includes in the category of “languages for specialized application areas,”

<table>
<thead>
<tr>
<th>Specialized Application Disciplines</th>
<th>Number of Languages Pre 1966</th>
<th>1-2</th>
<th>3-5</th>
<th>over 5</th>
<th>1966-1971</th>
<th>1-2</th>
<th>3-5</th>
<th>over 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>automated equipment checkout</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>civil engineering</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>computer assisted instruction</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical design (including simulation of digital computers)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>machine tool control</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movie animation</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>real time/process control</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>social science/ humanities systems programming</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Narrow Applications Across MultiDisciplines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>computer aided design</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>graphic and on-line display</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>query (excluding data base management systems)</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simulation—continuous (including block diagrams and analogue computers)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

all programming languages except those specifically intended for mathematical computations (both numeric and non-numeric, i.e., formula manipulation), business data processing, string and list processing, or any combinations of these. (The use of a language outside its intended application area does not cause it to be considered specialized, nor change the definition of its primary intended use.) This list of excluded categories is certainly debatable on the grounds of what it does and does not contain. The applications cited seem so fundamental to the use of computers that none should be considered as being specialized. Of the categories of specialization (discussed later) only simulation seems to have a reasonable potential for consideration as fundamental, and at this point in time such a conclusion does not seem justified to the author. Note also that systems programming is considered one of the specialized application areas; this view refutes the concept which has been expressed by some people that a language can only be considered good if it can be used to write its own compiler.

The terms “special-application-languages” and “application-oriented languages” will be used in this paper as synonyms for the phrase “languages for specialized application areas.”

Specific application areas and statistical trends

In Table II is a list of special application areas which have had programming languages developed for them. Some statistics are also given; the latter are presented in categories rather than specific numbers in order to show trends. It is virtually impossible to obtain a completely accurate count because of the existence of dialects, multigeneration languages (e.g., SIMSCRIPT I, I.5, II, II.5, and II Plus), and the differences between dates of definition, implementation, and actual usage. The distinction between pre-1966 and more recent years largely reflects existence on second generation computers. However, not all languages on second generation computers were implemented on third generation machines. The most complete lists of languages available—although they are by no means perfect or complete—are in References 15, 16, 17, 18. Discussions of almost all those languages developed by 1967 are in Reference 14, Chapter IX.

General statistical trends

As indicated just above, it is virtually impossible to obtain accurate statistics. This is due not only to the language generation problem but also to the fallibility
of any single human attempting to track this field. Until we get some scientific methodology for defining language generations, dialects, etc., and until a reliable reporting system is developed, the most we can hope for is some statistical trends. (See Reference 19, for a first attempt at dealing with the dialect problem.)

The trends that can be observed are all based on References 15, 16, 17, 18. While the figures cited have inaccuracies and in particular are based on date of inclusion of a language in the roster rather than its actual development, it seems fair to state that all the errors are probably consistent and hence relatively unbiased.

The first year in which any serious attempt was made to list "all" the special-application-oriented languages was 1968. In each of the four years 1968-1971 the number of special application languages was approximately 50 percent of the total number of languages listed. The latter were 82, 125, 139, and 164 in the four cited years. In 1969, 1970 and 1971 the total number of new languages added from the previous year was between 35 and 40 each year. In 1970 and 1971 the percentage of new special-application-languages was between 40 and 60 percent of all the new ones.* In both 1970 and 1971 the number of special-application-languages dropped from the roster (meaning clear non-use of the language) was about 50 percent of the total number of languages dropped. (The latter numbers are small, ranging between 5 and 10 percent of the total.)

These numbers are presented because of the trends they indicate, and certainly should not be used as definitive representations of the actual situation. For that reason these figures were deliberately not presented in a table or graph because that would imply more accuracy than is justified.

NEEDS TO BE MET IN DEVELOPING THE LANGUAGES

There are several needs which must be met by a new language or a modification of an existing one. The major needs are functional capability, a style suitable for the application area, and efficiency.

Functional capability

The inclusion of specific functional capability is probably the single most important need to be met by the application-oriented languages. Most of the deficiencies in a particular language actually reflect the omission of functional capabilities which a certain class of users need. For example, the FORTRAN user often wishes to do character manipulation, the COBOL user might wish to do floating point calculations, and the PL/I user might wish to do simulation. In considering functional capability, the three major issues are (a) the specific features which must be included (i.e., commands and data types), (b) the distinction between subroutines and a specific language facility, and (c) the facilities provided by the system. Each of these is now discussed in more detail.

Specific features

The specific features representing the functional capability of any language are the operations (i.e., commands) to be performed, and the data types. The kinds of operations to be provided depend completely on the application area, and of course must be consistent with the data types. As a very simple example, the existence of floating point numbers is extremely important in any language to be used for scientific computation, whereas they have little or no importance for business data processing. Therefore, the existence of a floating point or complex number data type is a functional requirement of many scientific problems, and naturally there must be executable statements which can operate on these data types. A less elementary example occurs in the civil engineering field where the ability to calculate the point of intersection of the tangents to two circles may be deemed important enough to incorporate directly into the language. Furthermore, a "point" with coordinates might be defined as a data type with operations provided to calculate areas. In the case of an application involving simulation, some indication of time and time-dependencies in both the data types and commands is absolutely crucial. In the case of a language developed for animated movies, the user must be able to address and manipulate pointers or markers on a screen. In the general area of graphics, a "graphical" data type is normally needed.

Subroutine versus language facility

Since all major languages have a subroutine capability of one kind or another, the question can legitimately be asked as to why the needs cannot simply be met through adding subroutines rather than either developing a new language or adding syntax and se-

* A similar figure for 1969 would be very misleading because the 1968 Roster did not contain any of the languages developed for CAI and their inclusion in 1969 perverts the statistics.
mantics to the existing language. It must be emphasized very strongly that there is a fundamental difference between functional capability (which can be represented either directly in the languages or via subroutines) and specific language facilities.

The easiest way to show this is by means of an example within the field of mathematics. (Since this was ruled out earlier as not being specialized, its use here is primarily to illustrate the concept of subroutines versus specific language facilities.) Suppose a person wished to do MATRIX addition in a language like FORTRAN. One could add a subroutine to perform those calculations and then invoke this facility by writing

\[ \text{CALL MATADD (A, B, C)} \]

This is fundamentally different in several ways from writing something of the following kind:

\[ \text{DECLARE A, B, C MATRICES} \]
\[ \text{C=A+B} \]

In the first instance, the user (meaning both the writer and reader of the program) has lost a certain mnemonic advantage and furthermore has the difficulty of remembering the types, sequence, and restrictions on the parameters that he is using. In this particular example, the main difficulty is to remember whether the stated call would provide A as the sum of B and C, or C as the sum of A and B; this is something which must be remembered or looked up in a manual. In the second example, what is happening is far clearer. In addition to the lack of "problem-oriented notation" inherent in using the CALL or equivalent statement, there are implementation difficulties and inefficiencies relative to linkages. In many cases, there is a great deal of extra code generated to invoke a subroutine, although the latter may in fact be quite small and would be more efficient placed in line. Unless the compiler is very sophisticated, it is likely to generate the linkage coding rather than placing the subroutine in line.

While subroutines do serve a very useful purpose in making additional functional capability available to the user, they should not be viewed as substitutes for additions to a language.

System provided facilities

Another aspect of functional capability is the built-in facility of the language and its associated compiler to do things which might need to be programmed in another language. For example, in the languages for computer assisted instruction, it is assumed that they will be used in an interactive mode and hence the language translators automatically provide for input and output from a terminal without the programmer having to specify much (if anything) about input/output. Furthermore, certain types of control flow are assumed by the system and handled automatically. In the case of graphic languages, the language (and its processor) automatically provide the necessary instructions to the graphics equipment.

Style suitable for application area

In the development of languages for special application areas, the style plays a major role, primarily because the users are normally not professional programmers. Style in a programming language has many facets, ranging from personal views on the importance (or non-importance) of blanks and punctuation, to the use (or non-use) of a GOTO command, to the selection of a particular word for getting input data (e.g., READ vs. GET). The major identifiable elements of style are vocabulary, personal preferences, and requirements affecting style.

Vocabulary

The second* most important need in an application-oriented language is the vocabulary or professional jargon that is involved. The whole raison d'être of many languages for specialized application areas is to allow the user to write his specialized jargon in a natural manner. Whereas he can certainly develop subroutines to accomplish what he wants, the ability to use his own terminology in a style natural to him is of paramount importance. It is normal that a person outside a particular area will find the nomenclature confusing or not understandable. All figures in this paper reflect this issue, i.e., the programs as written are generally not understandable to any reader outside the specific application area.

Personal preferences

It is unfortunate—but quite true—that in the development of one of these special-application-languages, the personal preferences of the developer have a significant effect, although they should of course be subordinated to the functional requirements and the vocabulary. For example, people who wish to have short data names and short statements in a language because

* The first is the functional capability.
they like that style may be forced into a different mode because the professional jargon of the field constantly uses long words. (In some systems, e.g., COGO, both short and long forms of key words are allowed to provide short cuts for experienced users.) The choice of a fairly rigid format versus a free form, or the selection of specific key words (e.g., READ versus GET) can sometimes make the difference between successful usage of the language versus constant unhappiness by the users. In some instances, people have strong views on punctuation and will change an existing language to eliminate (or include) punctuation in lieu of some other syntactic restriction. (It has been said facetiously that by choosing the correct subset of PL/I one merely has FORTRAN with semicolons.)

Background and past experience with specific equipment often have a strong personal influence. People who have used punched cards extensively tend to favor rigid formats with card columns used as syntactic delimiters. Those who have used on-line terminals generally tend to favor free form. Even in this latter case, there is considerable difference of opinion on the value of limiting each statement to the length of a single line.

The use of very simplistic styles, as illustrated in Figure 2, primarily consisting of a single key word at the beginning of a line, followed by some parameters, certainly forces one to reconsider the border line between programming languages and powerful macros. Certainly such a style is generally not "problem-oriented" which was described as one of the characteristics of a higher level language. However, such languages (e.g., COGO—see Figure 2) can be justified as higher level languages because of the distance of these operations from normal machine code. Thus, there is a large amount of information built into the compiler or the interpreter to perform specific functions which are directly related to an application (rather than merely enhancing normal machine code).

It should be emphasized that there is little or no way of determining which of two styles is better; in virtually every case it is a matter of individual taste, preference, previous usage, and jargon common to the application area.

Requirements affecting style

It should not be thought that all matters of style are arbitrary; some are influenced or forced by specific requirements of equipment—particularly the character set. In other cases, the intended use may affect the style of the language and environment will have a very specific effect. If the language is to be used in an inter-

Figure 2—COGO (civil engineering)

Table: Small COGO program for figure shown. In the figure above, given the coordinates of point 1, the length and azimuth (clockwise angle from north) of lines 1-7 and 1-95, the COGO program shown computes the coordinates of points 7 and 95 and the area of the triangle. In the program, the second line reads: Locate point 7 by going from point 1 a distance of 256.17 at an azimuth of 45 degrees 15 minutes 28 seconds.

STORE 1 1000. 2000.
LOCATE/AZIMUTH 7 1 256.17 45 15 28
LOCATE/AZIMUTH 95 1 350.00 102 35 12
AREA 1 7 95
PAUSE

Explanation:

Small COGO program for figure shown. In the figure above, given the coordinates of point 1, the length and azimuth (clockwise angle from north) of lines 1-7 and 1-95, the COGO program shown computes the coordinates of points 7 and 95 and the area of the triangle. In the program, the second line reads: Locate point 7 by going from point 1 a distance of 256.17 at an azimuth of 45 degrees 15 minutes 28 seconds.

program to a large number of persons long after the program is written.

**Efficiency**

It is obvious that one of the reasons for developing these application-oriented languages is efficiency. This involves the efficiency which comes both from ease of writing the programs and from suitable processors.

One major way in which an attempt is made to increase efficiency is to delete unwanted parts of a major language while (perhaps) adding on the functional capability (in the form of new language elements). While the person or group which has a large language available to them is under no obligation to use all of its facilities, it is perfectly clear that they are paying for the implementation of these unwanted features. The common and obvious practical solution to this problem is merely to have a compiler which implements a subset of a language, and this is frequently done. However, to the extent that the individual or group wants to clearly define a certain subset of syntax and semantics and give it a name, he has in effect defined a new language. It is not true that all languages can have subsets properly defined, if a program written in the subset is also to run on a compiler for the full language.

The ways in which suitable processors can be obtained are discussed in the next section.

**DESIGN PARAMETERS OF SPECIAL APPLICATION LANGUAGES**

The design parameters of the languages for special application areas essentially reflect the needs which were discussed in the previous section. Thus, the functional capability, a style suitable for the application area, and efficiency are clearly design parameters. However, there are two additional issues which were not discussed in the previous section but which are very significant in the development of these languages, namely the methods actually used to design and implement the languages. It is well-known that those issues are important in the design of any language, but they are perhaps more significant in the languages within the scope of this paper because the intended users are not professional programmers and hence are less likely to be tolerant of unnecessary idiosyncrasies. In more general languages, some of the idiosyncrasies are (unfortunately) forced into existence by the methodology used for language design and/or implementation. Finally, it is possible to summarize the potential language requirements.

**Methods of defining language**

There are three basic methods for defining a language. The first is the most obvious—namely, a straightforward definition. In this instance, the individual or group merely sits down and writes out a definition of the language. Depending upon their sophistication and the complexity of the language, they may use something like Backus Normal Form, or alternatively may simply use ordinary English and examples. A special case of this method involves making specific changes to an existing language, which may involve addition, deletion, changes, or all of these.

The second method of defining a language is through an extensible language system. This is an area which has become increasingly important over the past few years, although there is not much evidence of practical systems or significant usage as yet (see Reference 1). In this situation, the developer of the language for a special application area is limited in two ways. First, he is limited by the inherent style and capability of the base language, and second, he is constrained by the mechanism given to him to produce the extensions. If the extension facilities do not allow new data types to be added, then he is limited to the syntax and functional operations of new commands. For example, any macro system (e.g., PL/I) tends to allow new commands and/or new syntax to be developed but does not provide for new data types. Alternatively, other extensible systems, e.g., IMPs allow for both new commands and new data types, but do not allow for major changes in language style or format.

The third method of defining the language is via some system. In this case, which seems to be the most important and the most promising, the user or application programmer states his wishes to a person who can be defined as a "language designer" who then interacts with a system which produces relatively easily a language definition which meets the needs of the original user or programmer. While many people have talked about this for years, relatively little has actually been accomplished—see a later section. In the long run, it is clear to me that we must allow the user ample facilities to easily define and implement his own language subject to whatever constraints and quirks he may have. The key word here is "easily" and that is the major difficulty in achieving the general goal.

**Methods of implementation**

Just as there are several different ways of defining a language, so there are different broad techniques for
implementing them, and to some extent (but not en­
tirely) they match the methods of defining the lan­
guage. The first and most obvious method of imple­
mentation is a specific compiler or interpreter. This
would tend to be used most often in a case where a
language had been designed from scratch. Second,
paralleling exactly the use of extensible languages is the
extensible language compiler or equivalent facility.
This method might conceivably be used with a language
designed in another way, but it is highly unlikely to be
applicable. A third possibility is a very powerful macro
assembler which then allows the user quite a bit of
flexibility in terms of jargon, lists of parameters, etc.,
but gives him virtually no flexibility in style and overall
format. Finally, roughly corresponding to the user-
defined language via a system, is a system which gener­
ates a compiler or interpreter. This method of imple-
ment can be used even with the first case where
the individual has designed and defined the language
from scratch. The compiler generators that have been
the vogue for many years come close to satisfying this
requirement in theory, although none seem to do it in
practice. (See a later section.)

Potential language requirements

A brief summary of requirements (or considerations)
for potential desired language features is shown in
Table III. Obviously, any one language will only use
some of these. However, it is possible to find at least
one specialized language which requires or uses each of
these approaches or features.

SOME SPECIFIC APPLICATION AREAS
WITH EXAMPLES

Almost all special application areas tend to look
small and homogeneous when viewed from outside, but
large and filled with differing problems and issues when

qu Who discovered America?
aa Ericson
ab Leif Ericson
ty Your answer would be accepted by some.
ca Columbus
ty Yes
wa Ponce de Leon
ty No. He looked for the “Fountain of Youth.”
Try again.
un bl

Explanation:
Example uses the aa and ab operation codes. If the student
enters a response of “Ericson,” or “Leif Ericson,” the message
“You your answer would be accepted by some,” is typed to the
student. After the aa or ab match, the system continues to scan
statements until it finds the un statement. It then types the
contents of buffer 1. If the student responds with an answer
which does not match an aa, ab, ce, or wa statement, only the un
argument (the contents of buffer 1) is typed to the student. The
contents of buffer 1 might be, “Please try again.” Using a buffer
in this way saves the author from repeatedly entering the same
text for many un statements in the course.

Source: Reprinted by permission from p. 17, Coursewriter III for
System/360, Version 2, Application Description Manual,
Data Processing Division, H20-0587, © 1969 by
International Business Machines Corporation.

Figure 3—Coursewriter III
(computer assisted instruction)
viewed by those familiar with the field. A very good illustration of this can be seen merely by glancing at the papers on numerical control programming languages. Even though one language—namely APT—predominates, there are still enough technical issues surrounding its utility and implementation to cause the existence of numerous other languages. In particular, 32 others are listed.

In the field of computer-assisted instruction, there are over 30* languages utilizing different syntactic and conceptual approaches. Just glancing at Figures 3 and 4, for Coursewriter and FOIL, respectively, shows the wide variety of style, ranging from two-letter mnemonics to a PL/I-like language. A comparison of CAI languages can be found in Reference 23.

The situation in graphics is similar, but with another dimension of concept involved. In the computer-assisted instruction application area it can reasonably be argued that the application is unique and that there is little relevance to existing languages of somewhat wider purpose, e.g., FORTRAN, COBOL. (However, even in the CAI situation, the case can be made for the desirability of control statements, conditionals, and numeric calculations expressible by formulas as in FORTRAN.) In the field of graphic languages there is certainly a special facility that can be used by many applications (including, for example, computer-assisted instruction). The technical approaches and issues in graphics are as diverse as in other applications. In this case more of an argument can be made for providing the facilities as extensions to existing languages, e.g., GRAF, which is an extension of FORTRAN. However, most developers went to the other extreme with entire new languages, for example General Purpose Graphic Language. (See Figures 5 and 6, respectively.) Some languages take a middle ground by retaining some of the style of more popular languages, such as ALGOL or FORTRAN, but by no means accept compatibility with them as a constraint. (See for example Euler-G.) In each case the developer of the language was reflecting his view of how graphic information should be stored internally, and the most effective way in which it could be displayed and manipulated on a screen. In this application area, the physical environment plays a major role in the development of the language; thus the existence of lightpens, keyboards, push-buttons, etc., must be supported—if they are to be used by the graphic language.

SYSTEMS FOR DEVELOPING LANGUAGES FOR SPECIAL APPLICATION AREAS

It is unfortunate, but appears to be a fact, that there are no currently available systems which have actually been used in a practical way for the development of a significant number of languages (and their processors) for special application areas. It is not even

TY WOULD YOU LIKE TO CONTINUE THE EXERCISE?

ACCEPT
  IF 'NO', GO TO FINISH
  IF 'YES, OK'
    NUM = NUM + 1
    GO TO NEXT
  GO BACK PLEASE ANSWER YES OR NO

Explanation:
The TY causes the indicated typeout to be made. If the student responds with a NO there is a branch to a statement labeled FINISH. If either YES or OK are typed in the variable NUM has 1 added to it and control is transferred to the statement labeled NEXT. Any answer not including YES, NO, or OK causes the typeout from the system of PLEASE ANSWER YES OR NO and a return of control to the ACCEPT statement.


Figure 4—FOIL (computer assisted instruction)

* Not all of these are listed in Reference 18.
BEGIN WINDOW (A, B, C, D)
RECT (A, B, C, D)
AA=A+C/2
BB=B+D/2
LINE AA, B; AA, B+D
LINE A, BB; A+C, BB
END

Explanation:
A subroutine WINDOW is defined where A and B are the coordinates of one corner of a rectangle and C and D represent the horizontal and vertical dimensions. The subroutine to draw a rectangle is called and executed. The drawings of these windows are to have horizontal and vertical lines midway in the window so AA and BB compute the coordinates of the midpoints. The LINE commands cause the "midway" lines to be drawn.


Figure 6—General purpose graphic language

clear that any of the systems now in prototype stage will ever be satisfactory for that purpose. Virtually all of the systems known to the author have one or more of the following characteristics: (1) they require a compiler expert for their use; (2) they have been used to produce some minor variations on “normal” languages such as FORTRAN, ALGOL, etc.; (3) they are not really intended to be used to develop the types of languages discussed in this paper; (4) they give lip service—although little else—to the concept of allowing the average user to be able to define his own language and easily get a processor for it.

In theory, any compiler-compiler, meta-compiler or similarly designated system could be used for this purpose. However, there is a different emphasis in most of those developed to date. They have been designed primarily to provide an easy means of implementing known and widely used languages (e.g., FORTRAN, COBOL, ALGOL, PL/I) rather than as a tool for the development of new languages with uncommon requirements, and their processors. Thus the major considerations have pertained to efficiency of the resulting compiler, with an easy way to make minor changes in the syntax. A discussion of the past and potential use of such systems or translator writing systems in general is beyond the scope of this paper. A good survey is given in Reference 3.

Although no systems seem to have been widely, or even significantly, used for developing the types of languages within this paper, several have had limited use and/or have such intent for the future. A brief description of these will now be given.

(a) ICES
The Integrated Civil Engineering System (ICES) provides an overall system within which many language processors suitable for civil engineering can reside and use common facilities. There is also the capability of allowing the user to define new languages, or add facilities to one of the existing languages. This is done by means of the Command Definition Language (CDL). Although CDL has not been used very much in practice, at least one language, namely STRUDL, was developed using it. (A brief but relatively accessible summary of ICES, including CDL, is in Reference 14.)

(b) REL
The Rapidly Extensible Language (REL) System was (and is) intended for use by researchers in the fields of complex social and environmental processes.
It has a powerful English grammar, thus permitting individuals to communicate with the computer in a fairly “natural” language. In 1970 an experimental system was in operation on the IBM System 360/50 and was used to develop an animated film language and also by some social scientists.

(c) PLAN

The Problem Language ANalyzer (PLAN) has many facets to it, but the only one of interest in this context is its facility to allow the user to define simple new languages in an easy manner. A version providing graphics support allows the user to develop his language at an IBM 2250 and also provides him with many built-in graphics facilities.

(d) UAL

The User Adaptive Language (UAL) is another attempt to provide a user with the ability to dynamically create and modify a language in an interactive mode. This system provides the user with fairly sophisticated programming concepts (e.g., lists), but does not require him to use them.

(e) SDF

The Syntax Definition Facility (SDF) allows the user to define his language by means of illustrative sentences. The system indicates whether the input is ambiguous or contradictory to earlier information. By late 1971, it had been used primarily to implement the syntax of fairly standard language subunits, e.g., arithmetic and Boolean expressions.

(f) Extensible Languages

All extensible languages should—in theory—be usable for creating specialized languages. By late 1971, none seem to have been used in this way. See Reference 1.

BRIEF COMMENTS ON THE FUTURE

It seems unfortunate but true that the proliferation of higher level languages is likely to continue at about the same rate. The reason for this is that some of the causes and motivations behind the development of these languages rest in quirks of human nature rather than technological progress or lack thereof. Thus, as long as people find it fun to develop languages, as long as they want something which is specifically tailored exactly to their needs, and as long as they are going to find piecemeal faults with the existing languages, there is very little that technical progress can do to reduce the number of languages. On the other hand, there are some areas in which improved technology will have an enormous effect. For example, the existence of good extensible language systems, or good systems which can easily generate a language and its translator based on appropriate input from a language designer, will have a considerable effect. We might even envision specialized language generators, i.e., a system designed to allow the easy creation and implementation of languages in a single application area, e.g., CAI, graphics. ICES is a simple attempt in this direction for civil engineering.

In the opinion of this author, the ease and efficiency of using a language particularly suited to a specific application area is a desirable result which outweighs the disadvantages of the proliferation. A thorough discussion on the pressures and resources involved in the future development of these specialized languages is given in Reference 20.

SUMMARY

This paper has defined and discussed the class of languages which are designed for use in specialized application areas. These languages include about half of all higher level languages used in the United States at the time of this writing. A discussion of terminology showed why some of the commonly used terms for this class of languages are technically inappropriate.

The major needs to be met in developing these languages were shown to be functional capability, deletion of parts of an existing language, and a suitable style. Two specific application areas, namely CAI and graphics, were used to illustrate the existence of significantly different language styles within the same application area. Although there are a number of systems which purport to allow the user to easily define and implement his own language, and they are mentioned, none have actually been significantly used. A few brief comments on the future indicate that the proliferation serves a useful purpose and will continue.

REFERENCES

1 ACM SIGPLAN
   Proceedings of ACM SIGPLAN Conference on Extensible Languages
   SIGPLAN Notices Vol 6 No 12 December 1971
2 R S EANES
   An interactive syntax definition facility
   MIT Electronic Systems Laboratory Report ESL-R-397 September 1969
3 J FELDMAN D GRIES
   Translator writing systems
   Communications of the ACM Vol 11 No 2 February 1968
4 A HORMANN A LEAL D CRANDELL  
User adaptive language (UAL): A step toward man-machine synergism  
System Development Corporation Technical Memorandum TM 4539 June 1971
5 A HURWITZ J P CITRON J B YEATON  
GRAF: Graphic additions to FORTRAN  
Proceedings of the Fall Joint Computer Conference Vol 30 1967
6 Problem language analyzer (PLAN) program description manual  
IBM Corporation Form GH20-0594
7 PLAN graphics support for the IBM 2850 (application description manual)  
IBM Corporation Form H20-0535
8 E T IRONS  
Experience with an extensible language  
Communications of the ACM Vol 13 No 1 January 1970
9 H E KULSRUD  
A general purpose graphic language  
Communications of the ACM Vol 11 No 4 April 1968
10 W H P LESLIE Ed  
Numerical control programming languages  
North-Holland Publishing Company 1970
11 W E MANGOLD  
Status of NC language standardisation in I.S.O. (International Standards Organization)  
12 W M NEWMAN  
Display procedures  
Communications of the ACM Vol 14 No 10 October 1971
13 D ROOS Ed  
ICES system: General description  
MIT Dept of Civil Engineering R67–49 September 1967
14 J E SAMMET  
Programming languages: History and fundamentals  
Prentice-Hall Inc 1969
15 J E SAMMET  
Roster of programming languages, 1968  
Computers and Automation Vol 17 No 6 June 1968
16 J E SAMMET  
Roster of programming languages, 1969  
Computers and Automation Vol 18 No 7 June 1969
17 J E SAMMET  
Roster of programming languages, 1970  
Computers and Automation Vol 19 No 6B November 1970
18 J E SAMMET  
Roster of programming languages, 1971  
Computers and Automation Vol 20 No 6B June 1971
19 J E SAMMET  
Problems in, and a pragmatic approach to, programming language measurement  
Proceedings of the Fall Joint Computer Conference Vol 39 1971
20 F B THOMPSON et al  
REL: A rapidly extensible language system  
Proceedings of the 24th ACM National Conference 1969
21 F B THOMPSON B H DOSTERT  
The future of specialized languages  
Proceedings of the Spring Joint Computer Conference Vol 40 1972
22 R A WALTER  
A system for the generation of problem-oriented languages  
23 K L ZINN  
A comparative study of languages for programming interactive use of computers in instruction  
University of Michigan Center for Research on Learning and Teaching February 1969