AN OVERVIEW OF NONPROCEDURAL LANGUAGES

Burt M. Leavenworth
and
Jean E. Sammet

IBM Corporation

ABSTRACT

This paper attempts to describe some of the basic characteristics and issues involving the class of programming languages commonly referred to as "nonprocedural" or "very high level". The paper discusses major issues such as terminology, relativeness, and arbitrary sequencing. Five features of nonprocedural languages are described, and a number of specific languages are discussed briefly. A short history of the subject is included.

Section 1. INTRODUCTION

It is characteristic of the programming field that much time and energy has been, and will continue to be, spent on terminology. This is perhaps an indication of the youth as well as the rapid growth of the field. As an illustration of the confusion in which we find ourselves today, consider the following list of terms, each of which should be followed by the word "languages":

very high level
nonprocedural
less procedural
goal oriented
problem oriented
pattern directed
declarative
functional
relational
problem statement
problem definition
problem description
systems analysis
specification
result specification
task description

In addition to these terms, consider also the following:

automatic programming
artificial intelligence

This list is not necessarily a complete set of all the terms now being used by one or more groups of people to convey an intuitive notion of languages which in some sense are "higher" than FORTRAN, COBOL, PL/I, etc. The most common term used for this concept has been nonprocedural, and the most common phrase has been "what" rather than "how". That phrase refers to the (potential) facility of a user to indicate the goals (= "what") he wishes to achieve rather than the specific methods of solution (= "how") that must be used. As Feldman (1972, p. 15) points out, if these concepts could be carried out, then one could write

FIND INTEGERS A,B,C AND N SUCH THAT N > 2 AND
$A^N + B^N = C^N$

which is, of course, Fermat's last theorem.

While it is not possible to provide concrete definitions of these terms, this paper attempts to provide some general principles and characteristics pertaining to this class of languages. It is important to note that data definition languages are not included within the scope of this paper. Section 2 discusses some of the terminology and fundamental concepts of very high level languages. Section 3 provides highlights of the historical developments in this area. Section 4 indicates the major features of nonprocedural languages. Section 5 discusses various classes of languages, specifically business applications, nonsequencing, artificial intelligence, simulation and set oriented. Section 6 summarizes the key points of the paper.

Section 2. OVERALL VIEW OF TERMINOLOGY AND FUNDAMENTAL CONCEPTS

2.1 Introduction

It is not possible (for reasons to be given in Section 2.2) to assert that a given programming language is nonprocedural in the absolute sense, but one can state that it possesses certain nonprocedural features. What criteria or features can be used to perform this characterization? Before enumerating and discussing such criteria, it is well to review some fundamental properties of programs and programming languages. The remainder of this paper uses the terms "programming languages" and "languages" interchangeably.

In general, a program is a prescription for solving a particular problem. A procedure is a series of steps followed in a regular orderly definite way. Procedural programming is based to a great extent on the necessity to conform to the inherent sequential organization of the conventional (Von Neumann) digital computer. Therefore, a possible definition of a nonprocedural program is that it is a prescription for solving a problem without regard to any arbitrary sequencing requirements. (A discussion of what constitutes arbitrary sequencing and how it may be eliminated is given later.) In a broader context we will say that a nonprocedural program is a prescription for solving a problem without regard to details of how it is solved. That is, the solution should be specified implicitly in terms of structures or abstractions which are relevant to the problem rather than those operations, data and control structures which are convenient for some machine organization.

2.2 Relativeness

The most important point to note is that the term nonprocedural is definitively relative and changes as the state of the art changes.

The concept is elusive because nonproceduralness is not an absolute measurement but rather is an ever-moving target. Hence, it is better for everyone in the field to recognize this once and for all than to assume a fixed meaning for the term. In many ways the term "less procedural" is better because it makes very clear the relative nature of the concept. Note that "less procedural" involves a comparison of two languages whereas "nonprocedural" can apply to only one.

An examination of Figure 1 should make this clearer. A comparison of Figures 1(a) and 1(a*) shows the difference between assembly language and FORTRAN-like languages. Prior to the existence of FORTRAN, the expression A = (B + C) * D + E * F could be considered nonprocedural because it could not be written as one executable unit and translated by any program. Similarly, Figures 1(b) and 1(b*) indicate another level of relativeness since the FORTRAN program to do matrix multiplication can be handled by one statement in APL. Finally, the illustration of Figure 1(c) which is a program to CALCULATE THE SQUARE ROOT OF THE PRIME NUMBERS FROM 3 TO 95 AND PRINT IN 2 COLUMNS cannot be handled by any translating system known today. (It is essential to realize that the two forms shown in Figure 1(c*) are logically equivalent and the desirability of one form over the other (i.e., formal notation versus English) is a matter of personal preference.) The ability of a system to "understand English" is not at issue here; phrases that look like English may really depend on specific programming techniques, e.g., pattern matching and macro expansion, rather than English grammar. Putting the same point another way (because of its fundamental significance in this matter), it is entirely possible to design a formal language for doing mathematical problems in which the statement CALCULATE THE SQUARE ROOT OF THE PRIME NUMBERS FROM 3 TO 95 AND PRINT IN TWO COLUMNS is acceptable. At the other extreme, a natural and elegant looking phrase such as FIND X SUCH THAT $X^2 = 5$ is really equivalent to invoking a square root routine.

We actually have two types of relativeness: one involves the actual hardware and the other involves the problem or application area. In the case of the hardware we can only use as the base from which to measure, some particular machine or class of machines, e.g., the IBM System/360. As the machine changes, so does the relativeness. The reason that one must consider the machine is because certain terms or facilities which might be available in one machine are not in another. Thus, prior to the availability of floating point instructions in any hardware, the capability had to be included in the programming language, and thus would be considered higher level with respect to the machine. Once floating point became virtually universal on a computer it was removed from serious language consideration. We might consider now the concept of list processing. If computers commonly had such instructions built in, then they would not be needed in a so-called higher level language. Still another hardware example involves the parallel execution of many statements and there are currently varying degrees of availability of this hardware facility. Finally, the development of hardwired machines to implement languages which are high level by today's standards (e.g., SYMBOL(Chesley and Smith, 1971), and previously unimplemented proposals (e.g., FORTRAN (Bashkow, 1967) and ALGOL 60 (Anderson, 1961)) will require a major rethinking of this whole issue.

The relativity to the application area depends on one's knowledge and experience. For example, if a person knows only one method of integration or one method of solving differential equations, then a command which says INTEGRATE, or SOLVE this differential equation, may be considered to have a certain level relative to the knowledge of the user. A system like NAPSS (Synes, 1969) attempts to provide automatic numerical analysis and thus is very high level relative to the application. Similarly, the presence or absence of a SORT verb in a data processing language creates different levels of proceduralness.

The use of top-down programming can be considered an indication of relativeness while still using current languages. Thus, one uses successive pages and levels to indicate greater detail of portions of executable code.

One of the first instances in which it was realized -- albeit with great reluctance -- that nonproceduralness was really a relative term was by the CODASYL Language Structure Group (1962) which developed the Information Algebra. That work was originally viewed as being the development of a language which would be truly nonprocedural for data processing problems. The statement of the payroll program as given in the original report certainly looks on the surface as if it states only what is to be done rather than the details of how it is to be done. However, implicit even in a single equation of the IA is the notion of sequencing, i.e., some steps must be done before others. Since the amount of sequencing is a major characteristic in distinguishing levels of proceduralness, it is worth exploring this point a little further.

2.3 Sequencing

There is a difference between sequencing across statements and within one statement. The former requirements tend to be obvious in a problem. However, a single statement may or may not have sequencing imbedded in it, and it is not always obvious from looking at it. Sequencing is, of course, inherent in any mathematical expression that has precedence among the operators. Any data dependencies which are inherent in the problem statement also affect the sequencing by requiring the data to be obtained in the correct order. A trivial illustration of this is obvious by merely noting that one cannot produce outputs until after one has performed calculations on the inputs.

It is interesting to note that some of the side effects which occur in some programming languages are really due to differences in handling the sequencing of subexpressions within a single statement. For example, if A = B + DBL(B) where DBL is a built-in function which doubles and updates B, then the sequence of performing the operations will determine whether the original value of B or its double is used to calculate A; hence, the value of A will depend upon the sequence of computations on the right hand side.

As another illustration of the significance and relevance of sequencing, consider the problem statement shown in Figure 1(c*). This calculation could actually be performed in several ways. One way is to follow each number through the three "computations", i.e., test for primeness, and if the number is prime, then compute its square root and print it. However, depending upon the particular hardware and software, it might be more efficient to first determine all the primes, then calculate all the square roots of the identified primes, and then do all the printing. Obviously various combinations of these two extremes are also possible means of execution. Equally obviously, this is merely a prototype of a calculation involving a sequence of tasks each of which supplies data to the next but where each input datum is independent of the others. The program given in Figure 1(c) chooses only one of the alternatives; no discretion is left to the translator, whereas the statement in Figure 1(c*) could -- as indicated above -- be translated in several significantly different ways (e.g., using coroutines (Conway, 1963) which could have major impact on efficiency. The explicit sequencing used in the program of Figure 1(c) is not required for solution of the problem. One way of characterizing nonproceduralness is to say that the sequencing of any information by the programmer (except that which is inherent to the logic of the problem) is irrelevant.

2.4 Miscellaneous Terms

The term "automatic programming" in current terminology is at least a partial synonym for nonprocedural, but the former tends to encompass somewhat more. An attempt to indicate what is meant by automatic programming in today's environment was made by Balzer (1972). Of the many pages devoted to this subject, it seems most appropriate to quote the following sentences from page 10: "There is a continuum between the statement of a problem in terms of its initial state and its goal state and a specification of how to do it in a machine language. Most of computer language development can be viewed as a movement from specifying HOW to do something towards a statement of WHAT is desired."

The term "problem oriented" serves in part as a confusion factor. This term has frequently been used to mean those languages which were specifically designed for a narrow application area, e.g., civil engineering, CAI, equipment checkout. These languages -- which have more appropriately been called languages for specialized application areas by Sammet (1972) -- frequently increase the level of nonproceduralness by comparison with the broader based languages such as PL/I. However, this is primarily because the commands are directly related to the problem domain, e.g., DETERMINE AREA, TEST VOLTAGE. Thus the level of the primitive operations have an effect on the level of proceduralness. In this conceptual area, the difference between a very simple (albeit high level) programming language and application package is often blurred. A programming language contains a set of commands and has other features and facilities which can be applied flexibly within the problem domain. On the other hand, application packages generally (a) have fixed sets of routines and do not allow new ones to be added, (b) accept parameters as input, and (c) specify fixed file and data formats. Application packages usually have fixed sequences of routines but some allow the user to specify which routines are to be used and in what order.

There is actually an existing class of languages which represents a nontrivial level of nonproceduralness by today's standards -- namely the continuous and discrete simulation languages. In both instances the sequence in which statements are written is not necessarily the sequence in which they are executed. Nonproceduralness is also related to issues of parallelism and even an inherently procedural language such as PL/I allows the user to specify parallelism through the use of multitasking.

2.5 Need for a Metric

The need for metrics of some kind for measuring programming languages in general is discussed by Sammet (1971). For a given computer, and a specified level of knowledge about an application area, one would like to have a metric to indicate various levels of (high level) languages. For example, one might decide to consider machine code as having a measure of 1 and increase the number as the language became closer to the problem. One of the many difficulties in carrying out such a scheme is that different features in a language really have different levels of nonproceduralness.

Other ways of considering measures of nonproceduralness include specifying (a) the number of commands in the language and indicating which commands can actually be expressed in terms of others and hence are not basic; (b) the amount of information about the application domain that can or must be specified in the language; and (c) the amount of sequencing or parallelism that is involved.

Some people feel that it may be impossible to achieve any meaningful metric.

2.6 Language Versus Its Implementation

There is obviously a large difference between the language itself and the methodology or system which underlies or implements the language. For example, a particular system may actually provide several methods of executing a particular task (e.g., sorting, solution of differential equations). These methods may be imbedded in the system and the selection actually made by the system itself to provide the best solution based on an analysis of the parameters.

It is clear that a large part of the problem in implementing nonprocedural languages (or in automatic programming systems), pertains to efficiency. Thus, we could specify problems and their general solutions more easily if we were not so concerned about the efficient use of machine time. Clearly, as the economics continue to shrink machine costs and increase people costs, this will no longer be significant. On the other hand, a less procedural language might actually increase efficiency by allowing the compiler to optimize for a particular machine based on a higher level of intent.

2.7 Verification of Objectives

We do not currently -- and probably never will -- have any accurate way of ensuring that the person states exactly and precisely what he wishes to accomplish. The inaccuracy occurs at two levels -- one is the lack of a suitable language in which to express objectives and the other is the normal human tendency to be imprecise in knowing what is wanted. Relating this to current terminology, we need -- but don't have -- good assertion (= specification) languages. It is possible that techniques of proving programs correct could be used to design better specification languages. Note that at all levels of proving correctness the verification of the compiler itself is part of the problem.

What we have is a situation in which the user may intone the famous phrase, "Do as I say, not as I do". The paraphrase that would be appropriate for the computer field would be, "Do what I want you to do, not what I asked you to do".

Section 3. HISTORY

It is beyond the scope of this paper to give a detailed history of the terms and activities in this area, but it is appropriate to indicate a few highlights. In the very early stages of programming, i.e., in the first half of the 1950's, the phrase "automatic programming" was used to mean the process of writing a program in some higher level language. In that context, "higher level" was by comparison with machine code. As time went on, it became clear that the coding was only a portion of the entire problem solving task and therefore the phrase "automatic coding" came into use as meaning the use of a language such as FORTRAN. Thus, even in the very early days the proper distinction was made between coding (which is one aspect of the entire programming task) and the larger activity of programming. One of the first attempts at providing general information in a framework that might be called automatic programming was that by Young and Kent (1958). Using their work, and other inputs (e.g., SHARE Theory of Information Committee) the CODASYL Language Structure Group (1962) developed the Information Algebra. It is essentially a mathematically-oriented way of describing a data processing application in terms of the input-output relationships; these are actually defined by means of transformations on sets of entities called areas (which are analogous to files). The Information Algebra was used experimentally (Katz and McGee, 1963), but has never been developed to the point of significant implementation. Some extensions and modifications to these concepts were made by Lincoln (1971), Kobayashi (1972) and Morgenstern (1973).

A narrow approach, but one that is nevertheless in the spirit of less proceduralness, is exemplified in the early work on Report Program Generators and Decision Tables.

Various individuals and groups addressed parts of this problem. Homer (1966) proposed a system in which the user specified the input variables with their values, the functions involving the variables, and a list of output variables. The user did not need to supply this information in the correct sequence; the system was responsible for doing that and specific algorithms for achieving this were presented. It is not known to the authors whether this was ever implemented.

One of the ways to minimize the importance of sequencing is to allow the programmer to specifically designate which sets of statements can be done simultaneously. One of the earliest published suggestions for this is the set of "DO TOGETHER" and "HOLD" statements suggested by Opler (1965). A survey of other suggestions is given by Gosden (1966) which also contains a good bibliography on parallel processing from the viewpoint of language, operating systems, and proposed hardware.

By comparison with languages such as FORTRAN and COBOL, the string and pattern directed languages such as COMIT and SNOBOL contained less proceduralness.

Various experimental attempts at having the system do automatic numerical analysis include POSE (Schlesinger and Sashkin, 1967) and NAPSS (Synes, 1969).

In the late 1960's, a system called Absys 1 was implemented at the University of Aberdeen by Foster and Elcock (1969). This language was built on the concept of having the user specify assertions rather than commands. Thus, the user would write X = Y, Y = 2 in any order and the system would automatically assign the value of 2 to X. The system also causes programs to terminate unsuccessfully if the constraints are unsatisfiable.

Aside from these cited activities and some related concepts in artificial intelligence which are discussed in later sections, no other major developments took place. Some specific work started in the late 1960's and early 1970's and is discussed in Section 5. The major areas of current activity in automatic programming which are related to nonprocedural languages are the ISDOS Project at the University of Michigan and the Automatic Programing Project at MIT. (The Automatic Programming Laboratory at Harvard really deals with a different class of problems.)

Section 4. FEATURES OF NONPROCEDURAL LANGUAGES

4.1 Introduction

This section discusses 5 features which are deemed of major importance for inclusion in a programming language which purports to be nonprocedural, or which intends to lower the level of proceduralness. Some examples of languages possessing some of these features are given. (To some extent the material here overlaps slightly the discussion of terminology and characteristics in Sections 2 and 5, but this is unavoidable. This section concentrates on specific features, whereas Section 2 was emphasizing general characteristics and Section 5 emphasizes specific languages.)

4.2 Associative Referencing

We will use the term associative referencing to refer to the accessing of data based on some intrinsic property of the data. Associative referencing is usually provided in those languages that contain sets (see Section 5.6) as a data structure. The operation of selecting elements from previously defined sets, and of defining new sets from old based on some property of the members is sometimes called the "set former" (see, for example, SETL (Schwartz, 1973) and MADCAP (Morris and Wells, 1972)). The importance of associative referencing in nonprocedural languages is that the programmer does not have to specify access paths explicitly or program an algorithm to conduct a search for a specific data structure.

The LEAP language (Feldman and Rovner, 1969) is an important example of a conventional language (ALGOL) extended to accommodate sets and associative referencing. The paper is also interesting because it describes a hash-coded software referencing scheme which has been implemented. Language constructs were also described by Balzer (1967) for searching a "data collection" for members satisfying a given condition, but this proposal was never implemented to the authors' knowledge. Associative referencing is also contained in such disparate languages as SIMSCRIPT, STDS (Childs, 1968), TRAMP (Ash and Sibley, 1968), MacAIMS (Strnad, 1971), Codd's ALPHA language (Codd, 1971), SETL, and undoubtedly many more.

Codd (1972) defines algebraic operations on relations (Codd, 1970) which give a measure of the relative power of a language with respect to this type of data structure. In addition to the traditional set operations of Cartesian product, union, intersection, etc., he defines the relational operations of projection, join, division, and restriction. These operators (see Aggregate Operators below) effectively provide various types of associative referencing.

It is clear that associative referencing and the supporting data structures are important components of nonprocedural languages, and represent an area in which much work remains to be done.

4.3 Aggregate Operators

It is possible to avoid writing loops in some programming languages that provide aggregate operators. The polymorphic operators in APL are examples of operators that apply equally to scalars and aggregates, and that distribute over entire aggregates. In some cases the PL/I programmer can avoid loops by using certain operators which distribute over entire arrays or structures. There seems in general to be a close relationship between associative referencing and the aggregate operators we are discussing. It is certainly clear that the algebraic operators defined by Codd (see above) on relations are aggregate operators. It cannot be denied that the elimination of explicit sequencing by the means discussed above is truly a nonprocedural feature.

Two interesting aggregate operators which are useful in data processing applications are the bundle and glump operators proposed in the Information Algebra (CODASYL, 1962). The glump operator partitions an area (which is like a file) into subsets called elements such that an element contains all entities in the area having identical values for the given glump operator. The glump operation is used for grouping and summarization purposes, typical tasks in data processing. The bundle operator works on an ordered set of areas and, for each member of the Cartesian product, selects only those entities meeting a certain condition (such as equality of values in a particular domain).

The design and development of suitable aggregate operators is an important activity in nonprocedural languages.

4.4 Elimination of Arbitrary Sequencing

We will define arbitrary sequencing as any sequencing which is not dictated by the data dependencies of the application. In a functional program (see below), data dependencies are shown explicitly by the operator-operand structure of the program.

A functional programming language is one that does not contain either assignment or goto statements. As such, "functional" appears to be a synonym for "nonprocedural" since it is more involved with specifying the outcome desired as a function of the inputs, rather than indicating a step by step sequence of program steps. A program in a functional language such as pure LISP avoids side effects which are a concomitant of procedural programming. A side effect is caused in procedural languages by the modification of memory by the assignment statement. Pure functional languages produce no side effects since they have no assignment operation and cannot modify memory during expression evaluation.

Landin (1966) characterizes functional programs as a way of "describing things in terms of other things", and shows that this approach leads to the elimination of explicit sequencing. One example of functional programming would be APL "one liners" (without assignments, or without function calls with side effects). If a program satisfies the "single assignment" property (Tesler and Enea, 1968), (i.e., if no variable is assigned values by more than one statement), then the order of the statements is immaterial, and the correct program sequence can be determined by dependency analysis. It turns out that a single assignment program is really a functional program in disguise, where the intermediate results of computation are given explicit names. An interesting class of languages recently defined are the Red languages (Backus, 1972) which use no variables, no goto statements, and no built-in comprehension of recursively defined functions.

The ultimate expression of lack of arbitrary sequencing (perhaps we should use the term minimum sequencing) is a pure data flow programming language (see, for example, Kosinski, 1973). In this formalism, an application is decomposed into a set of modules which communicate with one another only at module interfaces. One module cannot consume a particular value until it is produced by another module, and conversely. The sequencing is governed strictly by data dependencies (see, for example, Figure 1(c)). This type of sequencing control can be provided by a coroutine structure (Conway, 1963). The best example of a well-known data flow programming language is GPSS (General Purpose Systems Simulator) in which sequencing of a simulation program is controlled by transactions (data) moving through the model.

4.5 Nondeterministic Programming and Parallelism

Facilities for nondeterministic programming appear in most of the artificial intelligence languages (see Section 5.4) and were inspired by Floyd (1967) who introduced new programming primitives for solving combinatorial problems. Essentially the primitives include the following:

(1) a multiple valued function called choice (n) whose values are the integers from 1 to n

(2) a success function, and

(3) a failure function.

The choice function causes a multiple branch in the execution of the program, each path being conceptually computed in parallel, with its own particular value of the choice as an argument. The success and failure functions label termination points of the computation. However, only those termination points labeled as success are considered to be computations of the algorithm. In other respects, a nondeterministic program using these primitives resembles a conventional program. In this approach, as described by Floyd, "a process with a very complicated control structure is represented by an algorithm with a simpler structure for an imaginary processor, and then converted to a more complicated algorithm for a conventional processor". This prescription describes a fundamental characteristic of nonprocedural programming. It must be emphasized here that the word "nondeterministic" used in the present context does not mean probabilistic but rather having a free choice.

In most cases, nondeterministic programs are executed as backtracking algorithms (Golomb and Baumert, 1965), and it is the hiding of the bookkeeping details of the backtracking (saving and restoring of the values of variables at branch points) that gives nondeterministic programming its nonprocedural flavor. Backtracking refers to a depth first tree search and is an important component of PLANNER (Hewitt, 1971) and other AI languages. Another method of implementing nondeterministic algorithms is called multiple-tracking (Irons, 1970) and refers to a breadth first search, which is useful in situations where the backtracking process does not terminate.

There is a close correspondence between nondeterministic programming and parallel programming in that the multiple paths of the choice function could be executed in parallel. The existence of inherent parallelism can be exhibited either naturally in a data flow language, or can be analyzed and detected in a more conventional programming language, such as FORTRAN IV (Russell, 1969 and Volansky, 1970). Use of statements such as FORK and JOIN (Anderson, 1965) provide the user with the facility to specify allowable parallelism, but simultaneously increase the amount of information he is supplying about the problem.

Communication and synchronization between concurrent processes is an important subject and has been extensively covered in the literature, but with primary emphasis on operating systems. A fundamental paper on this topic is by Dijkstra (1968).

Parallel processes appear in the multitasking facilities of PL/I, the collateral executions of ALGOL 68, the multiple paths in ECL (Cheatham and Wegbreit, 1972), and in the multiple processes of SAIL (Feldman, et al., 1972).

4.6 Pattern Directed Structures

The classical example of a pattern directed structure is given by Markov algorithms. A Markov algorithm consists of a set of replacement or substitution rules which are repeatedly applied to an input string of symbols. The sequencing algorithm is implicit in that the rules are always applied in a determined order. Each rule consists essentially of the directive: if a specified string is contained in the current input string, then replace it with a given string of symbols. Although replacement rules are, in general, order dependent, pattern matching is nonprocedural in the sense that its execution involves a complicated series of steps. The SNOBOL language is an extension and enhancement of the Markov algorithm idea where, however, the programmer is allowed to depart from the normal sequential control. The pattern directed string replacement rule in SNOBOL is nonprocedural in the sense that the more detailed specification of this process in a more "conventional" language would surely require a nondeterministic sequence of steps, or an even more complicated deterministic program.

The specification of a context free grammar in BNF is an example of a set of rewriting rules that are order independent. The CONVERT package (Guzman and McIntosh, 1966) is an example of the introduction of pattern matching facilities into LISP. A more recent extension to LISP emphasizing the role of pattern directed structures is embodied in the LISP 70 language developed at Stanford (Tesler et al., 1973).

Pattern directed structures are incorporated in the PLANNER language in a fundamental way. Specifically, PLANNER includes a pattern directed data base search, and the pattern directed invocation of procedures. The pattern directed data base search allows the user to ask for data items called assertions which match a given pattern, while pattern directed procedure invocation has the capability to initiate tasks of the form "call a subroutine which will achieve the desired result at this point" (Sussman, et al., 1971). It is noteworthy that program monitoring may be considered to be a generalization of pattern directed invocation of procedures where the pattern is being continuously matched against the object to be monitored (Fisher, 1970).

Pattern matching is intimately associated with nondeterministic programming, but is probably less procedural than the latter. However, it may be that to solve more complicated problems, the programmer has to give more hints to the program, possibly in procedural form. It is clear that pattern matching is related to, and conceptually similar to, associative referencing of data.

Section 5. SPECIFIC LANGUAGES

5.1. Introduction

There are so many languages which exhibit some of the major characteristics of nonprocedural languages that it is not possible to describe them all. Furthermore, since they tend to fall into groupings based on either application area or technical approach, they are being discussed in such groups. It is essential to realize that the distinction between a "language" and a "language system" becomes very fuzzy when dealing with these developments; the reader is cautioned to understand that the "languages" mentioned may in reality be a part of a larger system which is providing significant support for the language itself.

5.2 Business Applications Languages

One of the areas in which considerable work has been done to develop very high level languages is the area commonly called business data processing. Depending upon the group doing the work, the language(s) may be developed either for this whole application area or specific portions thereof. In addition to all the other difficulties pointed out in earlier sections, there is another continuum which makes it difficult to make clear-cut distinctions. At one end is a specific query, while at the other end is an implicit calculation in which the system must essentially carry out an unspecified sequence of computations and/or retrievals to obtain the desired result. In order to make this continuum more obvious, consider the following statements in a restricted form of English:

a) FIND ALL PEOPLE IN DEPARTMENT A WHO MAKE MORE THAN $B.

b) FIND THE AVERAGE SALARY OF ALL PEOPLE IN DEPARTMENT A WHO MAKE MORE THAN $B.

In the first case this is a standard query which can be answered on most data base management systems. However, if only the second statement is given, there is a great deal of unspecified processing required because the system must first find all the relevant people, then apply the concept of "average", and then perform the computation. Thus, general data base management systems themselves have many of the characteristics of very high level languages.

While it is beyond the scope of this paper to discuss the issue of communicating with the computer in English, it is nevertheless true that this approach is being taken by most groups working in this area. This tends to be done in three different ways -- (i) by a restricted formal language which looks like English; (ii) by a set of imperative statements or questions in "natural English"; and (iii) by a questionnaire from the computer to the person wherein the responses are considerably limited but still in English.

Both (ii) and (iii) are objectives of the MIT Automatic Programming Group under W. Martin which is developing several levels of language within the framework of a system called Protosystem I (Project MAC, 1973). One of these is MAPL, which is a language for building relational models of the world. The latter is considered to be made up of a collection of objects which are divided into subsets and where the concept "a kind of" is allowed. A significant part of their approach is the use of a questionnaire which is carefully designed for a particular specialized application area. The user will answer the questions, and also communicate more information about the details of his problem in a "natural English". From this information the system will translate to successively lower levels of language for eventual execution. It is expected that this process will be iterative in the sense that the user may be asked for further details after an initial processing of the questionnaire information. Furthermore, the user will be given the opportunity to determine that the program generated by the system is what he really wants; this is deemed essential because the user is specifying the "what" and the system is determining the "how".

The ISDOS project under D. Teichroew and A. Merten at the University of Michigan emphasizes what they refer to as "problem statement languages" (which are essentially the same as what Sammet calls "problem defining languages"). As with various other conceptually similar approaches, they wish to distinguish carefully between the specifications and requirements of the problem on one hand, and the method of achieving these on the other. For example, a requirement might be that pay checks should be printed in alphabetical order, but this is quite different from having a user specify "SORT" as a statement in a program. The latter might be unnecessary as a command if other aspects of the system design caused an alphabetical sequence to occur automatically.

In the framework of the ISDOS project, it is expected that users will write a problem description in PSL (Problem Statement Language) and this will be analyzed by the PSA (Problem Statement Analyzer). PSL is designed to provide the Problem Definer (i.e., today's systems analyst) with a better method of stating the requirements for an information processing system. A careful distinction is made between the latter, and the programs which implement these requirements. Stating requirements is done in the current version being developed, namely PSL/II (Hershey, et al., 1973), which allows for 8 sections in the problem statements. The analyst writes one or more of these sections in a top-down fashion providing more detail at lower levels. The intermediate level of having the system make design decisions on file formats and structure of programs was being worked on in mid-1973. Relating this to Section 4, extensions to the Problem Statement Language will probably provide some associative reference and aggregate operators and presumably will concentrate heavily on eliminating arbitrary sequencing.

Other languages in this same general area are mentioned or described in survey papers by Teichroew (1972) and Couger (1973) and will not be repeated here. The major ones mentioned are ADS, Hoskyns System, Information Albegra, PSL, and TAG. The key concepts of the

Information Algebra were stated in Sections 3 and 4 of this paper. ADS (Lynch, 1969) and TAG (IBM) basically consist of a set of forms describing an entire application which are filled out by the user or system analyst and then machine-analyzed for use by programmers and file designers. If the forms could be translated to working programs, then we would actually have an extension of RPG concepts from a single program to a whole set of programs, i.e., a full application or system. According to Couger (1973), "the Hoskyns system accepts system specifications and converts them to COBOL programs without manual intervention".

5.3 "Nonsequencing" Languages

As stated in Section 4.4, one of the key characteristics of a "less procedural" language is to minimize the amount of sequencing specified by the programmer. In particular, it is desirable to be able to distinguish between the order of evaluation and the ordering of statements in the source program. While many of the specific languages discussed in Section 5 have this characteristic, the languages mentioned here tend to emphasize (directly or indirectly) this facility.

Continuous simulation languages (e.g., CSSL (SCi, 1967), CSMP (IBM)) have had this capability for years. As a very simple example, the user might write the following equations, where T is an independent variable:

```
X = R cos A
R = 50.0
Z = W T
W = 5.0
```

The compiler would automatically rearrange these to be

```
W = 5.0
Z = W T
R = 50.0
X = R cos A
```

While this is quite trivial and could be done equally easily by the user, in large simulations involving many interrelated equations with many variables, the rearrangement is laborious and error prone and can be done more easily by a computer.

Another relevant aspect of the continuous simulation languages is that they are frequently used to model analog computers which have many computations occurring simultaneously. To handle this digitally (and hence, sequentially), the compiler must automatically cause certain computations to be temporarily deferred; this is most noticeable in dealing with the (numerical) solution of several differential equations.

One of the most significant of the more general languages with this approach is the interactive language ABSET being developed at the Computer Research Group at the University of Aberdeen, Scotland. (See Elcock, 1971.) A key element in their language is the use of sets in which the user is allowed to say "this is true for all members of this set" and the notions of TRUE and FALSE are appropriately applied. A simple illustration of the application of the latter together with a deduction from the stated assertions is the following:

```
A + B = 3 AND A = 1;
```

From this the interpreter deduces that B = 2 since the meaning of AND requires that the two halves of the first statement are both true.

A proposed -- but unimplemented -- approach to this is Compel (Compute Parallel) described by Tesler and Enea (1968). They propose that all variables in each statement belong to two mutually exclusive groups: Input, Output. All rules on statement sequencing are replaced by the rule "The statement that outputs variable A must be executed before every statement that either inputs A or inputs some B such that B depends on A".

5.4 Artificial Intelligence Languages

Mention has been made before of the distinction between stating what is to be done and how to do it. In stating what is to be done, an individual is in some sense describing the problem, or stating what results are required. When we discuss methods of solving problems, we may be in the domain of artificial intelligence, which cannot be discussed here for lack of space. However, languages which have been developed in the AI field have features which allow the user to think about and specify his problem in a nonprocedural way.

The PLANNER language is important and will be emphasized here because it is the archetype of a class of nonprocedural and "problem solving" languages in the artificial intelligence field. PLANNER is a mixture of programming and theorem proving techniques and, as stated earlier, incorporates three basic ideas:

automatic backtracking
pattern directed data base search
pattern directed invocation of procedures

Backtracking has been mentioned previously in Section 4.5.

Pattern directed data base search is a generalization of associative referencing and allows the user to ask for data items called assertions in the data base which match a given pattern.

Pattern directed invocation of procedures allows a procedure to be invoked not by its name but by the function that it performs. It also allows a set of recommendations to be specified which controls the pattern match algorithm so that alternatives are tried in a particular order. Pattern directed invocation really "constitutes a new view of programming based not on the traditional hierarchical organization of a set of subroutines, but rather on a set of cooperating asynchronous modules". (Balzer, 1972).

CONNIVER (Sussman and McDermott, 1972) is an extension and modification to PLANNER in which automatic backtracking is eliminated. It is argued that automatic backtracking almost forces the user to regard all his problem solving methods as independent. That is, backtracking provides a mechanism for generating alternatives and for erasing the consequences of an alternative which is later found to be untenable. CONNIVER provides local environments or contexts for each alternative to which changes can be made. CONNIVER is an attempt to exploit all the "good" ideas in PLANNER by providing some primitives originally hidden in PLANNER, and by concentrating more on the programming aspects of the language rather than on the theorem proving orientation.

The QA4 language (Derksen, et al., 1972) which is based on QA3 (Green, 1969) is similar in many respects to PLANNER (for example, heavy reliance on pattern matching and built-in backtracking). However, QA4 relies more on the use of sets, whereas PLANNER would implement the same features by using more complex procedures. QA4 also has the concept of "bags" which are like sets except that they allow several instances of the same element; this simplifies many arithmetic problems. QA4 also has a context mechanism which allows the storing of conditional plans (sequences of operators) under different contexts or alternative contingencies.

SAIL (Feldman, et al., 1972) is a dialect of ALGOL 60 and was developed primarily with the Stanford Hand-Eye System in mind. It is based on the associative processing features of LEAP but contains many new features such as backtracking and matching procedures. Matching procedures are somewhat similar to IPL-V "generators" and are used to generate strategies, and for coding of complex associative contexts. SAIL has sets and lists as data structures, and a new data type called a "context" which is useful for state saving and backup. SAIL contains multiple processes and follows the important principle that an occurrence in one process can influence the flow of control in other processes.

REF-ARF (Fikes, 1970) consists of a nondeterministic language for stating problems and a processor that attempts to find a successful execution of the nondeterministic program.

GOL (Pople, 1972) is a LISP extension for nondeterministic programming and is similar to PLANNER in some respects, except that GOL uses semantic methods to generate its state space, rather than syntactic, or deductive methods.

The following systems, while more properly described as application programs than languages, are included because they illustrate problem solving features which might be useful in nonprocedural programming languages.

The STRIPS system (Fikes and Nilsson, 1971) is a problem solving program that attempts to find a sequence of operators that transform a given initial model (configuration) into a model in which a given goal formula is true. STRIPS represents a model as a collection of formulas in the first-order predicate calculus, and uses a resolution approach (Robinson, 1965) to theorem proving in order to answer questions about the model.

Another system, called PROW (Waldinger and Lee, 1969), generates programs from descriptions of their inputs and outputs in the predicate calculus and also uses a resolution theorem prover. A similar use of the predicate calculus as a programming language, but not using an automatic theorem prover, is illustrated by Manna and Waldinger (1971).

DENDRAL (Feigenbaum, et al., 1971), which analyzes mass spectrograph data, is interesting because it demonstrates the effective incorporation of domain specific information into an artificial intelligence environment.

5.5 Simulation Languages

Simulation languages introduced nonprocedural concepts in essentially two areas: associative referencing and the concept of a "process". Although associative referencing was not a particularly new idea at the time, SIMSCRIPT introduced the concept of representing and modeling systems in terms of entities and their attributes, and the referencing and updating of data items based on indirect addressing of their properties rather than on an explicit search. Both LISP and IPL-V used the notion of property lists but did not exploit this idea in quite the same way as SIMSCRIPT.

GPSS has been attractive to nonprogrammers because of its block diagram, or flow graph, orientation and the fact that sequencing of transactions was determined by properties of the data and state configurations of the system, rather than by explicit control flow specification in more conventional programming languages. The process concept, as typified by GPSS and by the original SIMULA (Dahl and Nygaard, 1966), is a method of discretely simulating concurrency in the real world, and is nonprocedural because the details of the sequencing and scheduling of processes is hidden from the programmer. This style of programming, using primitives such as ACTIVATE, HOLD, WAIT and TERMINATE (in SIMULA) has been called quasi-parallel programming (Knuth, 1968).

Continuous simulation languages generally involve resequencing of equations specified by the user and are illustrated in Section 5.3.

5.6 Set Oriented Languages

The traditional aggregate data structures in programming languages have been arrays where the concepts of indexing and sequencing have been important. Languages which provide sets as data structures do not rely on the relative position of data items in the sets for accessing purposes, but make use of associative referencing as discussed earlier.

We will discuss the languages SETL (Schwartz, 1973) and MADCAP (Morris and Wells, 1972) as representative of a class of set oriented langauges. (ABSET, which also has set operations, was mentioned in Section 5.3.) SETL is a very high level mathematically oriented language. Its important composite data structures are finite unordered sets, tuples, and functions. The set operations in both languages are very similar except that SETL allows heterogeneous sets. Functions in both SETL and MADCAP are not only available in the conventional sense but can also be represented by sets of tuples, i.e., relations.

Both languages have a "set former" capability which is to say that they provide associative referencing on the elements of sets. SETL has a "compound operator" which works very much like the APL reduction operator, and both languages have other constructions which can be used to obviate loops in most cases.

MADCAP has a backtracking facility (not currently provided in SETL) as well as a control structure called an iterative expression.

As an example of the power of SETL, consider the following expression which specifies the prime numbers between 2 and 100:

```
{P,2 <= P <= 100 ↑ (∀ 2 <= N < P ↑ (P//N)NE. 0)}
```

which can be read in English as "the set of P's between 2 and 100 such that for every N greater than or equal to 2 and less than P the remainder of P/N is not equal to zero".

The above specification is obviously not an efficient one; a practical program at the very least would just consider the odd numbers from 3 to 100.

Earley (1973) proposes four data structures which are very similar to those used in SETL and MADCAP:

- Tuples (fixed collections of heterogeneous objects which can be accessed by name)
- Sets (unordered non-repeating collections of objects)
- Relations (sets of tuples)
- Sequences (ordered collections of objects)

Earley's major point is that these structures allow a relational level of description in which data structures may be described in terms of essential relationships between the data items, ignoring the particular access paths between them. This prescription clearly states one of our principles of nonprocedural programming.

Section 6. SUMMARY

This paper has attempted to describe some of the basic characteristics and issues arising from the term "nonprocedural languages". It has been emphasized repeatedly that this is a relative term that changes as the state of the art changes. Numerous other terms are used almost synonymously for this concept. Among the major issues discussed were the relativeness of the concept from several viewpoints, the importance of minimizing unnecessary sequencing in writing programs or defining problems, the need for a metric to measure levels of nonproceduralness, the relation between the language and its implementation, the relationship to work on verifying the objectives of the user, and the connection with the field of artificial intelligence.

A brief history of highlights in this field was given. A discussion of the following 5 features which should be included in languages purporting to be nonprocedural was given: associative referencing, aggregate operators, the elimination of arbitrary sequencing, nondeterministic programming and parallelism, and pattern directed structures.

A number of specific languages were discussed very briefly under the general categories of business applications, "nonsequencing", artificial intelligence, simulation, and set oriented.

FIGURE 1

```
          (a)                                (a*)
CLA B                             A = (B + C) * D + E * F
ADD C
MPY D
STO T
CLA E
MPY F
ADD T
STO A

          (b)                                (b*)
DO I = 1, M                       A ← B + .x C
DO J = 1, N
C (I, J) = 0
DO K = 1, P
C(I,J) = C(I,J) + A(I,K) * B(K,J)

          (c)                                (c*)
DO I = 3 TO 95 BY 2           PRINT (2), SQ(PRIME (3,95))
IF PRIME (I)
  THEN PUT SKIP LIST                       or
  (I, SQRT (I));
  ELSE RETURN;
END;                          CALCULATE THE SQUARE ROOT
                                OF THE PRIME NUMBERS
                                FROM 3 TO 95 AND
                                PRINT IN 2 COLUMNS
```

REFERENCES

Anderson, J.P., "A Computer for Direct Execution of Algorithmic Languages", Proceedings EJCC, Vol. 20 (1961).

Anderson, J.P., "Program Structures for Parallel Processing", CACM, Vol. 8, No. 12 (Dec. 1965).

Ash, W. and Sibley, E.H., "TRAMP, an Interpretive Associative Processor With Deductive Capabilities", Proceedings ACM 23rd National Conference (1968).

Backus, J., Reduction Languages and Variable-Free Programming, IBM Research Report RJ 1010, Yorktown Heights, N.Y. (April 1972).

Balzer, R.M., "Dataless Programming", Proceedings FJCC, Vol. 31 (1967).

Balzer, Robert, Automatic Programming, Technical Memo, Information Sciences Institute, University of Southern California, (September 1972).

Bashkow, T.R., Sasson, A., and Kronfeld, A., "A System Design of a FORTRAN Machine", IEEE Trans. Elec. Comp., Vol. EC-16, No. 4 (August 1967).

Cheatham, T.E. Jr. and Wegbreit, Ben, "A Laboratory for the Study of Automating Programming", Proceedings SJCC, Vol. 40 (1972).

Chesley, G.D. and Smith, W.R., "The Hardware-Implemented High-Level Machine Language for SYMBOL", Proceedings SJCC, Vol. 38 (1971).

Childs, D.L., "Description of a Set-Theoretic Data Structure", Proceedings FJCC, Vol. 33, Part 1 (1968).

CODASYL Language Structure Group, "An Information Algebra Phase I Report", CACM, Vol. 5, No. 4 (April 1962).

Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM, Vol. 13, No. 6 (June 1970).

Codd, E.F., A Data Base Sublanguage Founded on the Relational Calculus, Report RJ 893, IBM Research Laboratory, San Jose, Calif. (July 1971).

Codd, E.F., Relational Completeness of Data Base Sublanguages, Report RJ 987, IBM Research Laboratory, San Jose, Calif. (March 1972).

Conway, M.E., "Design of a Separable Transition-Diagram Compiler", CACM, Vol. 6, No. 7 (July 1963).

Couger, J.D., "Evolution of Business System Analysis Techniques", Computing Surveys, Vol. 5, No. 3 (Sept. 1973).

Dahl, Ole-Johan and Nygaard, Kristen, "SIMULA- an ALGOL-Based Simulation Language", CACM, Vol. 9, No. 9 (Sept. 1966).

Derksen, Jan, Rulifson, John F., and Waldinger, Richard J., "The QA4 Language Applied to Robot Planning", Proceedings FJCC, Vol. 41 (1972).

Dijkstra, E.W., "Cooperating Sequential Processes" in Programming Languages, (F. Genuys, ed.), Academic Press, New York (1968).

Earley, Jay, Relational Level Data Structures for Programming Languages, Computer Science Dept., University of California, Berkeley (March 1973) (submitted for publication in Acta Informatica).

Elcock, E.W. et al., "Abset, a Programming Language Based on Sets: Motivation and Examples" in Machine Intelligence 6 (B. Meltzer and D. Michie, eds.), American Elsevier (1971).

Feigenbaum, E., Buchannan, B. and Lederberg, J., "Generality and Problem Solving: A Case Study Using the DENDRAL Program" in Machine Intelligence 6 (B. Meltzer and D. Michie, eds.), American Elsevier (1971).

Feldman, J.A., Automatic Programming, Report No. CS-255, Stanford University Computer Science Dept. (Feb. 1972).

Feldman, J.A., Low, J.R., Swinehart, D.C., and Taylor, R.H., "Recent Developments in SAIL - An ALGOL-Based Language for Artificial Intelligence", Proceedings FJCC, Vol. 41 (1972).

Feldman, J.A. and Rovner, P.D., "An ALGOL-Based Associative Language", CACM, Vol. 12, No. 8 (Aug. 1969).

Fikes, R., "Ref-Arf: A System for Solving Problems Stated as Procedures", Artificial Intelligence, Vol. 1, No. 1 (1970).

Fikes, Richard E. and Nilsson, Nils J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence, Vol. 2 (1971).

Fisher, David A., Control Structures for Programming Languages, Ph.D. Thesis, Carnegie Mellon University (May 1970).

Floyd, R.W., "Nondeterministic Algorithms", JACM, Vol. 14 (Oct. 1967).

Foster, J.M. and Elcock, E.W., "ABSYS 1: An Incremental Compiler for Assertions: An Introduction" in Machine Intelligence 4 (B. Meltzer and D. Michie, eds.), American Elsevier (1969).

Golomb, S. and Baumert, L., "Backtrack Programming", JACM, Vol. 12, No. 4 (Oct. 1965).

Gosden, J.A., "Explicit Parallel Processing Description and Control in PRograms for Multi- And Uni-Processor Computers", Proceedings FJCC, Vol. 29 (1966).

Green, C., "Theorem-Proving by Resolution as a Basis for Question-Answering Systems" in Machine Intelligence 4 (B. Meltzer and D. Michie, eds), American Elsevier (1969).

Guzman, Adolfo and McIntosh, Harold V., "CONVERT", CACM Vol. 9, No. 8 (Aug. 1966).

Hershey, E.A. et al., PSL/II Language Specifications, Version 1.0, ISDOS Working Paper No. 68, University of Michigan, Dept. of Industrial and Operations Engineering, Ann Arbor, Michigan (Feb. 1973).

Hewitt, Carl, "PLANNER: A Language for Proving Theorems in Robots", Proceedings IJCAI (1969).

Hewitt, Carl, "Procedural Embedding of Knowledge in PLANNER", Proceedings IJCAI 2 (Sept. 1971).

Homer, E.D., "An Algorithm for Selecting and Sequencing Statements as a Basis for a Problem-Oriented Programming System", Proceedings 21st National ACM Conference (1966).

Irons, Edgar T., Multiple-Track Programming, Research Report NO. 70-1, Department of Computer Science, Yale University (1970).

IBM. System/360 Continuous System Modeling Program: User's Manual GH 20-0367.

IBM. The Time Automatic Grid System (TAG): Sales and Systems Guide GY20-0358 (reprinted in J.F. Kelly Computerized Management Information Systems, MacMillan (1970)).

Katz, J. and McGee, W.C., "An Experiment in Non-Procedural Programming", Proceedings FJCC, Vol. 24 (1963).

Knuth, Donald E., The Art of Programming, Addison-Wesley, Reading Mass. (1968).

Kobayashi, Isamu, "An Algebraic Model of Information Structure and Information Processing", Proceedings ACM National Conference (1972).

Kosinski, Paul R., A Data Flow Programming Language, Report RC 4264, IBM Research Laboratory, Yorktown Heights, N.Y. (March 1973).

Landin, P.J., "The Next 700 Programming Languages", CACM, Vol. 9, No. 3 (March 1966).

Lincoln, N., "Languages for Parallel Thinkers", 1971 (unpublished; distributed at ACM SIGPLAN Symposium on Languages for Systems Implementation (Oct. 1971)).

Lynch, H.J., "ADS: A Technique in System Documentation", SIGBDP Database, ACM Headquarters, Vol. 1, No. 1 (Spring 1969).

Manna, Z. and Waldinger, R., "Towards Automatic Program Synthesis", CACM, Vol. 14, No. 3 (March 1971).

Morgenstern, M., Automating the Design and Optimization of Information Processing Systems, AUTOMATIC PROGRAMMING GROUP Internal Memo 10, Project MAC, MIT, Cambridge, Mass. (Feb. 1973).

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

Opler, A., "Procedure Oriented Language Statements to Facilitate Parallel Processing", CACM, Vol. 8, No. 5 (May 1965).

Pople, Harry E., Jr., "A Goal-Oriented Language for the Computer" in Representation and Meaning - Experiments With Information Processing Systems (H.A. Simon and L. Siklossy, eds.), Prentice-Hall, Englewood Cliffs, N.J. (1972).

Project MAC Annual Progress Report No. 10, Automatic Programming Division, MIT, Cambridge, Mass. (to be published Dec. 1973).

Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle", JACM, Vol. 12, No. 1 (Jan. 1965).

Russell, E.C., Automatic Program Analysis, Ph.D. Dissertation, Dept. of Electrical Engineering, UCLA (1969).

Sammet, J.E., "Problems In, and a Pragmatic Approach to Programming Language Measurement", Proceedings FJCC, Vol. 39 (1971).

Sammet, J.E., "An Overview of Programming Languages for Specialized Application Areas", Proceedings SJCC, Vol. 40 (1972).

Schlesinger, S., and Sashkin, L., "POSE: A Language for Posing Problems to a Computer", CACM, Vol. 10, No. 5 (May 1967).

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

SCi Simulation Software Committee, "The SCi Continuous System Simulation Language (CSSL)", Simulation, Vol. 9, No. 6 (Dec. 1967).

Strnad, A.L., "The Relational Approach to the Management of Data Bases", Proceedings IFIP Congress (1971).

Sussman, G.J., Winograd, T., and Charniak, E., MICRO-PLANNER Reference Manual, MIT AI Memo 203A (Dec. 1971).

Sussman, Gerald J. and McDermott, Drew V., "From PLANNER to CONNIVER: A Genetic Approach", Proceedings FJCC, Vol. 41 (1972).

Synes, L.R. and Roman, R.V., Syntactic and Semantic Description of the Numerical Analysis Programming Language (NAPSS), CSD TR 11 (Revised), Purdue University, Computer Science Dept., Lafayette, Indiana (Sept. 1969).

Teichroew, Daniel, "A Survey of Languages for Stating Requirements for Computer-Based Information Systems", Proceedings FJCC, Vol. 41 (1972).

Tesler, L.G. and Enea, H.J., "A Language Design for Concurrent Processes", Proceedings SJCC, Vol. 32 (1968).

Tesler L. Enea, H.J. and Smith, D.C., "The LISP 70 Pattern Matching System", Proceedings IJCAI 3 (1973).

Volansky, S.A., Graph Model Analysis and Implementation of Computational Sequences, Ph.D. Dissertation, Dept. of Electrical Engineering, UCLA (1970).

Waldinger, Richard J. and Lee, Richard C.T., "PROW: A Step Toward Automatic Program Writing", Proceedings IJCAI (1969).

Young, J.W. Jr. and Kent, H.K., "Abstract Formulation of Data Processing Problems", J. Industrial Engineering (Nov.-Dec. 1958).