PROGRAMMING LANGUAGES—THE FIRST 25 YEARS*

P. WEGNER

Abstract—The programming language field is certainly one of the most important subfields of computer science. It is rich in concepts, theories, and practical developments. The present paper attempts to trace the 25 year development of programming languages by means of a sequence of 30 milestones (languages and concepts) listed in more or less historical order. The first 13 milestones (M1–M13) are largely concerned with specific programming languages of the 1950's and 1960's such as Fortran, Algol 60, Cobol, Lisp, and Snobol 4. The next ten milestones (M14–M23) relate to concepts and theories in the programming language field such as formal language theory, language definition, program verification, semantics and abstraction. The remaining milestones (M24–M30) relate to the software engineering methodology of the 1970's and include a discussion of structured programming and the life cycle concept. This discussion of programming language development is far from complete and there are both practical developments such as special purpose languages and theoretical topics such as the lambda calculus which are not adequately covered. However, it is hoped that the discussion covers the principal concepts and languages in a reasonably nontrivial way and that it captures the sense of excitement and the enormous variety of activity that was characteristic of the programming language field during its first 25 years.

Index Terms—Abstraction, assemblers, Algol, axioms, Cobol, compilers, Fortran, Lisp, modularity, programming languages, semantics, structures programming, syntax, verification.

I. Three Phases of Programming Language Development

THE 25 year development of programming languages may be characterized by three phases corresponding roughly to the 1950's, 1960's, and 1970's. The 1950's were concerned primarily with the *discovery* and *description* of programming language concepts. The 1960's were concerned primarily with the *elaboration* and *analysis* of concepts developed in the 1950's. The 1970's were concerned with the development of an effective software *technology*. As pointed out in [96], the 1950's emphasized the *empirical* approach to the study of programming language concepts, the 1960's emphasized a *mathematical* approach in its attempts to develop theories and generalizations of concepts developed in the 1950's, and the 1970's emphasized an *engineering* approach in its attempt to harness concepts and theories for the development of software technology.

*Reprinted from *IEEE Transactions on Computers*, Dec. 1976, 1207–1225.

The author is with the Division of Applied Mathematics, Brown University, Providence, RI 02912.

Manuscript received September 3, 1976; revised August 23, 1976. This work was supported in part by the AFOST, the ARO, and the ONR under Contract N00014-76-C-0160.

1950–1960 Discovery and Description

A remarkably large number of the basic concepts of programming languages had been discovered and implemented by 1960. This period includes the development of symbolic assembly languages, macro-assembly languages, Fortran, Algol 60, Cobol, IPL V, Lisp, and Comit [72]. It includes the discovery of many of the basic implementation techniques such as symbol table construction and look-up techniques for assemblers and macro-assemblers, the stack algorithm for evaluating arithmetic expressions, the activation record stack with display technique for keeping track of accessible identifiers during execution of block structure languages, and marking algorithms for garbage collection in languages such as IPL V and Lisp.

This period was one of discovery and description of programming languages and implementation techniques. Programming languages were regarded solely as tools for facilitating the specification of programs rather than as interesting objects of study in their own right. The development of models, abstractions, and theories concerning programming languages was largely a phenomenon of the 1960's.

1961–1969 Elaboration and Analysis

The 1960's were a period of elaboration of programming languages developed in the 1950's and of analysis for the purpose of constructing models and theories of programming languages.

The languages developed in the 1960's include Jovial, PL/I, Simula 67, Algol 68, and Snobol 4. These languages are, each in a different way, elaborations of languages developed in the 1950's. For example, PL/I is an attempt to combine the "good" features of Fortran, Algol, Cobol, and Lisp into a single language. Algol 68 is an attempt to generalize, as systematically and clearly as possible, the language features of Algol 60. Both the attempt to achieve greater richness by synthesis of existing features and the attempt to achieve greater richness by generalization have led to excessively elaborate languages. We have learned that in order to achieve flexibility and power of expression in programming languages we must pay the price of greater complexity. In the 1970's there is a tendency to retrench towards simpler languages like Pascal, even at the price of restricting flexibility and power of expression.

Theoretical work in the 1960's includes many of the basic results of formal languages and automata theory with

applications to parsing and compiling [1]. It includes the development of theories of operational and mathematical semantics, of language definition techniques, and of several frameworks for modeling the compilation and execution process [26]. It includes the development of the basic ideas of program correctness and program verification [54].

Although much of the theoretical work started in the 1960's continued into the 1970's, the emphasis on theoretical research as an end in itself is essentially a phenomenon of the 1960's. In the 1970's theoretical research in areas such as program verification is increasingly motivated by practical technological considerations rather than by the "pure research" objective of advancing our understanding independently of any practical payoff.

In the programming language field the pure research of the 1960's tended to emphasize the study of abstract structures such as the lambda calculus or complex structures such as Algol 68. In the 1970's this emphasis on abstraction and elaboration is gradually being replaced by an emphasis on methodologies aimed at improving the technology of programming.

1970–? Technology

During the 1970's emphasis shifted away from "pure research" towards practical management of the environment, not only in computer science but also in other scientific areas. Decreasing hardware costs and increasingly complex software projects created a "complexity barrier" in software development which caused the management of software-hardware complexity to become the primary practical problem in computer science. Research was directed away from the development of powerful new programming languages and general theories of programming language structure towards the development of tools and methodologies for controlling the complexity, cost, and reliability of large programs.

Research emphasized methodologies such as structured programming, module design and specification, and program verification [41]. Attempts to design verifiable languages which support structured programming and modularity are currently being made. Pascal, Clu, Alphard, Modula, and Euclid are examples of such "methodology-oriented languages."

The technological, methodology-oriented approach to language design results in a very different view of what is important in programming language research. Whereas work in the 1960's was aimed at increasing expressive power, work in the 1970's is aimed at constraining expressive power so as to allow better management of the process of constructing large programs from their components. It remains to be seen whether the management of software complexity can be substantially improved by imposing structure, modularity, and verifiability constraints on program construction.

II. Milestones, Languages, and Concepts

The body of this paper outlines in greater detail some of the principal milestones of programming language development. The milestones include the development of specific programming languages, and the development of implementation techniques, concepts and theories.

The four most important milestones are probably the following ones.

Fortran, which provided an existence proof for higher level languages, and is still one of the most widely used programming languages.

Algol 60, whose clean design and specification served as an inspiration for the development of a discipline of programming languages.

Cobol, which pioneered the development of data description facilities, was adopted as a required language on department of defense computers and has become the most widely used language of the 1970's.

Lisp, whose unique blend of simplicity and power have caused it to become both the most widely used language in artificial intelligence and the starting point for the development of a mathematical theory of computation.

We shall consider about 30 milestones, and use this section as a vehicle for presenting a brief history of the programming language field. The milestones can be split into three groups. Milestones M1–M13 are concerned largely with specific programming languages developed during the 1950's and 1960's. Milestones M14–M23 consider certain conceptual and theoretical programming language notions. Milestones M24–M30 are concerned with programming languages and methodology of the 1970's.

M1—The EDVAC report, 1944 [81]: This report, written by Von Neumann in September 1944, contains the first description of the stored program computers, subsequently called Von Neumann machines. It develops a (one address) machine language for such computers and some examples of programs in this machine language.

M2—Book by Wilkes, Wheeler, and Gill, 1951 [83]: This is the first book on both application software and system software. It discusses subroutines and subroutine linkage, and develops subroutines for a number of applications. It contains a set of "initial orders" which act like a sophisticated loader, performing decimal to binary conversion for operation codes and addresses, and having relative addressing facilities. Thus, the basic idea of using the computer to translate user specified instructions into a considerably different internal representation was already firmly established by 1951.

M3—The development of assemblers, 1950–1960: The term "assembler" was introduced by Wilkes, Wheeler, and Gill [83] to denote a program which assembles a master program with several subroutines into a single run-time program. The meaning of the term was subsequently narrowed to denote a program which translates from symbolic machine language (with symbolic instruction codes and addresses) into an internal machine representation. Early assemblers include Soap, developed for the IBM 650 in the mid 1950's and Sap developed for the IBM 704 in the late 1950's.

The principal phases of the assembly process are as follows:

1) scanning of input text;
2) construction of symbolic address symbol table;

3) transliteration of symbolic instruction and address codes;
4) code generation.

The first assemblers were among the most complex and ingeneous programs of their day. However, during the 1960's the writing of assemblers was transformed from an art into a science, so that an assembler may now be regarded as a "simple" program. The development of an implementation technology for assemblers was an essential prerequisite to the development of an implementation technology for compilers.

M4—Macro assemblers, 1955–1965: Macro-assemblers allow the user to define "macro-instructions" by means of macro-definitions and to call them by means of macro-calls. A macro-facility is effectively a language extension mechanism which allows the user to introduce new language forms (macro-calls) and to define the "meaning" of each new language form by a macro-definition.

A macro-assembler may be implemented by generalizing phases 2 and 3 of the previously discussed assembly process. Phase 2 is generalized by construction of an additional symbol table for macro-definitions. Phase 3 is generalized by requiring table look-up not only for symbolic instruction and address codes but also for macro-calls. The table look-up process for macro-calls is no longer simple transliteration, since the determination of a macro-value may involve parameter substitution and nested macro-calls. However, the implementation technology for macro-assemblers may be regarded as an extension and generalization of the implementation technology for assemblers. The seminal paper on macro-assemblers is the paper by McIlroy [54]. A discussion of implementation technology for macro-assemblers is given in [84].

Macro-systems may be generalized by relaxing restrictions on the form of the text generated as a result of a macro-call. Macro-systems which allow the "value" of a macro-call to be an arbitrary string (as opposed to a sequence of machine language instructions) are called macro-generators. Trac [55] is an interesting example of a macro-generator.

Macro-systems may be generalized even further by generalizing the permitted syntax of macro-calls. Waite's Limp system [85] and Leavenworth's syntax macros [52] are early examples of such generalized macro-systems. Macro-systems of this kind are useful for implementing language preprocessors which translate statement forms and abbreviations of an "extended language" into a "strict language" which generally has a smaller vocabulary but is more verbose.

Generalized macro-systems may be implemented by macro-definition tables which are constructed and used in precisely the same way as for macro-assemblers. Generalized macro "values" require more general macro-body specifications in the macro-definition table while more general syntax for macro-calls requires a more sophisticated scanner for recognizing macro-calls in the source language text.

Assembly and macro-languages have been discussed in some detail because they illustrate how a simple language idea (the idea of transliteration) backed up by a simple implementation mechanism (the symbol table) leads to a class of simple languages (symbolic assembly languages) and how progressive generalization of the language idea together with a corresponding generalization of the implementation technology leads to progressively more complex classes of languages. This example is useful also because it illustrates how the language and implementation mechanism for assemblers are related to the language and implementation mechanisms for compilers.

M5—Fortran, 1954–1958 [27]: Fortran is perhaps the single most important milestone in the development of programming languages. It was developed at a time of considerable scepticism concerning the compile-time and run-time efficiency of higher level languages, and its successful implementation provided an existence proof for both the feasibility and the viability of higher level languages. Important language concepts introduced by Fortran include:

variables, expressions and statements (arithmetic and Boolean);
arrays whose maximum size is known at compile-time;
iterative and conditional branching control structures;
independently compiled (nonrecursive) subroutines;
COMMON and EQUIVALENCE statements for data sharing;
FORMAT directed input-output.

Advances of implementation technology developed in connection with Fortran include the stack model of arithmetic expression evaluation.

Fortran was designed around a model of implementation in which run-time storage requirements for programs, data and working storage was known at compile-time so that relative addresses of entities in all subroutines and COMMON data blocks could be assigned at compile-time and converted to absolute addresses at load time.

This model of implementation required the exclusion from the language of arrays with dynamic bounds and recursive subroutines. Thus, Fortran illustrates the principle that the model of implementation in the mind of the language designers may strongly affect the design of the language. Although Fortran is machine independent in the sense that it is independent of the assembly level instruction set of a specific computer, it is machine dependent in the sense that its design is dependent on a virtual machine that constitutes the model of implementation in the mind of the programming language designer.

M6—Algol 60, 1957–1960: Whereas Fortran is the most important practical milestone in programming language development, Algol 60 is perhaps the most important conceptual milestone. Its defining document, known as the Algol report [62], presents a method of language definition which is an enormous advance over previous definition techniques and allows us for the first time to think of a language as an object of study rather than as a tool in problem solution. Language syntax is defined by a variant of the notation of context-free grammars known as Backus–Naur Form (BNF). The semantics of each syntactic

language construct is characterized by an English language description of the execution time effect of the construct.

The Algol report generated a great deal of sometimes heated debate concerning obscurities, ambiguities and trouble spots in the language specification. The revised report [63] corrected many of the less controversial anomalies of the original report. Knuth's 1967 paper on "The remaining trouble spots of Algol 60" [44] illustrates the nature of this great programming language debate. The participants in the debate were at first called Algol lawyers and later called Algol theologians.

Important language constructs introduced by Algol 60 include:

block structure;
explicit type declaration for variables;
scope rules for local variables;
dynamic as opposed to static lifetimes for variables;
nested if-then-else expressions and statements;
call by value and call by name for procedure parameters;
recursive subroutines;
arrays with dynamic bounds.

Algol 60 is carefully designed around a model of implementation in which storage allocation for expression evaluation, block entry and exit and procedure entry and exit can be performed in a single run-time stack. Dijkstra developed an implementation of Algol 60 as early as the fall of 1960 based on this simple model of implementation [23]. However, this semantic model of implementation was implicit rather than explicit in the Algol report. Failure to understand the model led to a widespread view that Algol 60 required a high price in run-time overhead, and to an exaggerated view of the difficulty of implementing Algol 60. An explicit account of the model of implementation is given in [69].

Algol 60 is a good example of a language which becomes semantically very simple if we have the right model of implementation but appears to be semantically complex if we have the wrong model of implementation. The model of implementation is more permissive than Fortran with regard to run-time storage allocation, and can handle arrays with dynamic bounds and recursive procedures. However, it cannot handle certain other language features such as assignment of pointers to pointer valued variables and procedures which return procedures as their result. These language features are accordingly excluded from Algol 60, illustrating again the influence of the model of implementation on the source language.

The Algol 60 notion of block structure quickly became the accepted canonical programming language design folklore and, in spite of its merits, excercised an inhibiting influence on programming language designers during the 1960's. Viewed from the vantage point of the 1970's it appears that nested scope rules for accessibility of identifiers and nested lifetime rules for existence of data structures may be too restrictive a basis for specifying modules and module interconnections in programming languages of the future. Alternatives to block structures are discussed in the sections on Simula 67, Snobol 4, and APL.

M7—Cobol 61, 1959–1961 [12]: Cobol represents the culmination and synthesis of several different projects for the development of business data processing languages, the first of which (flowmatic) was started in the early 1950's by Hopper. See [72] for an account of this development. Important language constructs introduced by Cobol include:

explicit distinction between identification division, environment division, data division, and procedure division;
natural language style of programming;
record data structures;
file description and manipulation facilities.

The two principal contributions of Cobol are its natural language programming style and its greater emphasis on data description. Natural language programming style makes programs more readable (by executives) but does not enhance writability or the ability to find errors. It constitutes a cosmetic change of syntax, sometimes referred to as "syntactic sugaring." It has not been widely adopted in subsequent programming languages but may possibly come into its own if and when the use of computers becomes commonplace in the home and in other nontechnical environments.

The contribution of Cobol to programming language development is probably greater in the area of data description than in the area of natural language programming. By introducing an explicit data division for data description to parallel a procedure division for procedure description Cobol factors out the data description problem as being of equal importance and visibility as the procedure description problem.

The significance of Cobol was greatly enhanced when it was chosen as a required language on DOD computers. Cobol was one of the earliest languages to be standardized, and has provided valuable experience (both positive and negative) concerning the creation and maintenance of programming language standards. It is currently used by more programmers than any other programming language.

Why is it that Cobol, in spite of certain defects in its procedure division, has become the most widely used language among commercial, industrial and government programmers? One reason is perhaps that standardization carries with it advantages that make the use of an imperfect standard more desirable than a more perfect but possibly more volatile alternative. Another perhaps more important reason may be that the advantages of Cobol's powerful facilities in its data division outweigh its imperfections in the procedure division, making it more suitable than languages like Fortran in the large number of medium and large scale data processing applications in business, industry and government. Cobol was behind the state of the art in its procedure division facilities but ahead of the state of the art in its data division facilities. The attractiveness of a language for data processing problems does not appear to depend as critically on its procedure description facilities as on its data description facilities.

M8—PL/I, 1964–1969 [67]: Fortran, Algol 60, and Cobol 61 may be regarded as the three principal first generation higher level languages, while PL/I and Algol 68 may be

regarded as the two principal second generation higher level languages. PL/I was developed as a synthesis of Fortran, Algol 60, and Cobol, taking over its expression and statement syntax from Fortran, block structure and type declaration from Algol 60, and data description facilities from Cobol. Additional language features include the following:

programmer defined exception conditions (the ON statement);
based variables (pointers and list processing);
static, automatic and controlled storage;
external (independently compiled) procedures;
multitasking.

PL/I illustrates both the advantages and the problems of developing a rich general purpose language by synthesis of features of existing languages. One of the lessons learned was that greater richness and power of expression led to greater complexity both in language definition and in language use. PL/I is a language in which programming is relatively easy once the language has been mastered, but in which verifiability and subsequent readability of programs may present a problem. Any language definition of PL/I is so complex that its use for the informal or formal verification of correctness for specific programs is intractable.

M9—Algol 68, 1963–1969 [82]: Whereas PL/I was developed by synthesis of the features of a number of existing languages, Algol 68 was developed by systematic generalization of the features of a single language, namely Algol 60. The language contains a relatively small number of "orthogonal" language concepts. The power of the language is obtained by minimizing the restrictions on how features of the language may be combined. Interesting language features of Algol 68 include:

a powerful mechanism for building up composite modes from the five primitive modes *int, real, bool, char, format*;
identity declarations;
pointer values, structures, etc;
carefully designed coercion from one mode to another;
a parallel programming facility.

The generality of the language can be illustrated by considering the mode (type) mechanism. Composite modes can be built up from modes *m,n* by the mode construction operators [ ] *m* (multiples), *struct* (*m,n*) (structures), *proc* (*m*)*n* (procedures), *ref m* (references) and *union* (*m,n*) (unions). Any mode constructed in this way may itself be the "operand" of a further mode construction operator as in *struct* ([ ] *ref m, proc* (*ref ref m*) *ref n*). Thus, an infinite number of different modes can be constructed from the primitive ones. Each definable mode has a set of values which must be manipulatable by the assignment operator and other applicable operators. Procedures may have any definable mode as a parameter, so that there must be provision for passing of parameters in any definable mode. The above discussion illustrates how generality in Algol 68 is obtained by starting from a small set of orthogonal concepts (the primitive modes and mode construction operators) and generating a very rich class of objects (modes and mode values) by simply removing all restrictions on the manner of composition.

The defining document for Algol 68 (Algol 68 report) [80], is an important example of a high quality language definition, using a powerful syntactic notation for expressing syntax and semiformal English for expressing semantics. However, the report introduces its own syntactic and semantic terminology and can be read only after a considerable investment of time and effort. The reader must become familiar with syntactic terms such as "notion," "metanotion," and "protonotion," and with semantic terms such as "elaboration," "unit," "closed clause," and "identity declaration."

Algol 68 has not been widely accepted by the programming language community in part because of the lack of adequate implementation and user manuals. However, an ultimately more important reason appears to be that the language constructs of Algol 68 are too general and flexible to be readily assimilated and used by the applications programmer.

M10—Simula 67 1965–1967 [17]: Simula 67 is a milestone in the development of programming languages because it contains an important generalization of the notion of a block, which is called a *class*. A Simula class, just like an Algol block, consists of a set of procedure and data declarations followed by a sequence of executable statements enclosed in begin-end parentheses. However, Simula has a "class" data type and allows the assignment of instances of classes to class-valued variables. Whereas local procedures and data structures of a block are created on entry to the block and disappear on exit from a block, local objects of a class (declared in its outer block) remain in existence independently of whether the class body is being executed as long as the variable to which the instance of the class has been assigned as a value remains in existence.

Classes may function as *coroutines* with interleaved execution of executable instructions of two or more class bodies. Execution of a command "resume C_2" in class C_1 causes the current state of execution of C_1 to be saved followed by transfer of control to the current point of execution of C_2.

The separation between class creation and class execution allows data structures in a class to endure between instances of execution and makes the class more useful than the block as a modeling tool for inventory control systems, operating system modules, data types and other entities which may be characterized by a data structure representing the "current state" and a set of operations for querying and updating the current state.

The usefulness of classes in modeling is enhanced even further by Simula conventions concerning the accessibility of local procedure and data declarations in a class.

If an instance of the class C has been assigned to the variable X, then the local identifier I of this instance of the class C can be accessed as $X \cdot I$. The ability to access local identifiers of a class in this way has both advantages (direct access to class attributes) and disadvantages (not enough control over restricting communication between system modules).

The *subclass* mechanism of Simula 67 allows the procedure and data declarations of a class C to become part of the environment of the class B by means of the declaration "C *class* B." If we think of the procedure and data declarations of a class as its set of attributes, then "C *class* B" causes B to have all the attributes of C plus any additional attributes local to B. B is called a subclass of C since it is the subset of C which has the attributes of B in addition to those of C.

The subclass mechanism is a very effective language extension mechanism. It has been used by the Simula 67 designers to design hierarchies of environments for Simula 67 users. Perhaps the best known of these environments is the simulation environment, which is created by first creating a list processing class containing a set of useful list processing procedures, and then defining a subclass simulation which uses list processing procedures to implement simulation primitives. Thus, Simula is not inherently a simulation language but merely a language which may easily be adapted to simulation by language extension.

A Simula class is a better primitive module for modeling objects or concepts than the Algol procedure because of its ability to remember its data state between instances of execution. It has been used as a starting point for the development of a notion of modularity appropriate to modular programming languages of the 1970's.

M11—IPL V, 1954–1958 [65]: IPL V is a list processing language developed specifically for the solutions of problems in artificial intelligence. It was widely used in the 1950's and 1960's for the programming artificial intelligence applications in areas such as chess, automatic theorem proving and general problem solving.

IPL V has primitive instructions for creating and manipulating list data structures. It is an assembly level list processing language with a 1 + 1 address code (the first address names an operand and the second address names the next instruction). Both programs and data are represented by lists. There are a number of system cells with reserved names, such as a communication cell for communicating system parameters, a subroutine call stack and a free storage list cell. A large number (over 100) of system defined subroutines (processes) are available to aid the user. The semantics of IPL V instructions is specified by defining an instruction interpreter for IPL V instructions.

IPL V was an important milestone both because it was widely used for a period of over ten years by an important segment of the artificial intelligence community and because it pioneered many of the basic concepts of list processing. For example, the notion of a free storage list serving as a source for storage allocation and as a sink to which cells no longer needed are returned was pioneered in IPL V. IPL V may well have been the first language to define its instructions by a software specified instruction execution cycle (virtual machine).

M12—Lisp, 1959–1960 [56]: Lisp, like IPL V, was developed for the solution of problems in artificial intelligence. However, Lisp may be thought of as a higher level (as opposed to machine level) programming language.

Lisp has two primitive data types referred to as lists and atoms. It has the following simple but powerful set of primitive operations.

A *constructor* $cons[x;y]$ for constructing a composite list from components x and y.

Two *selectors* $car[x]$, $cdr[x]$ for, respectively, selecting the first component and remainder of the list.

Two predicates $atom[x]$, $eq[x;y]$ which, respectively, test whether x is an atom and whether two atoms x and y are identical.

A compound conditional of the form $[p_1 \rightarrow a_1; p_2 \rightarrow a_2; \cdots; p_n \rightarrow a_n]$ which may be read as "*if* p_1 *then* a_1 *else if* p_2 then $a_2 \cdots$ else if p_n then a_n" and results in execution of the action a_i corresponding to the first true predicate p_i. Binding operators $lambda[x;f]$ and $label[x;f]$ which bind free instances of x in f so that they, respectively, denote function arguments and recursive function calls.

The set of primitive Lisp operations have been enumerated explicitly because they exhibit in the simplest terms the essential operators in a nonnumerical processing language. Every nonnumerical processing language must contain constructors for constructing composite structures from their components, selectors for selecting components of composite structures and predicates which permit conditional branching determined by the "value" of the arguments. The compound conditional is a very attractive control structure which was first developed for Lisp and later incorporated into Algol 60. The Lisp binding operators (lambda and label) provide a mechanism for handling functions which have functions as arguments as in the lambda calculus.

Lisp is sufficiently simple to permit the development of a relatively tractable mathematical model. McCarthy used this model as a starting point for the development of a mathematical theory of computation [57]. He considered many of the basic theoretical programming language issues such as mathematical semantics, proofs of program correctness (including compiler correctness) and proofs of program equivalence (by recursion induction) several years before they were considered by anyone else.

McCarthy also developed a definition of Lisp by means of a Lisp interpreter (the APPLY function) which, given an arbitrary Lisp program P with its data D executes the program P with data D. The Lisp APPLY function demonstrated as early as 1960 the technique of defining a programming language L by an interpreter written either in L or in some language definition language. It became the starting point for the subsequent development of theories of operational semantics [95], and for the development of interpreter based language definition languages such as VDL [49].

Lisp contributed a great deal to our understanding of programming language theory. It is also the most influential and widely used artificial intelligence language, its popularity being due in no small measure to its unique blend of simplicity and power. Lisp is certainly among the most important milestones in the development of programming languages.

M13—Snobol 4 1962–1967 [32]: During the 1950's it was felt that mechanical translation and other glamorous

language understanding tasks could be greatly facilitated by the development of string manipulation languages with special purpose linguistic transformation aids. The Comit language [99] was developed for this purpose during the period 1957–1961. Comit was a good linguists language with many special purpose linguistic transformation features, but is not a clean programming language because it does not have string-valued variables to which strings may be assigned as values. The deficiencies of Comit led to the development of Snobol 4 during 1962–1967.

Snobol 4 has data values of the type—integer, real, string and pattern, as well as programmer defined data types. However, Snobol 4 has no block structure or declarations. A given variable, say X, may take on string values, numerical values or pattern values at different points of execution. The data type is carried along as part of the Snobol 4 data value and is checked dynamically at execution time to determine whether it is compatible with the operation that is to be applied to it. Dynamic type checking runs counter to the philosophy of static type checking in conventional block structure languages. It introduces additional run-time overhead and increases the proportion of programming errors that will not be discovered until execution time. However, introduction of block structure and explicit type declarations into Snobol 4 would totally change its character, and it is not clear that such a change would be for the better.

The most important programming language contribution of Snobol 4 is the pattern data type. A pattern is an ordered (finite or infinite) set of strings (and string attributes). Snobol 4 has pattern construction operators for constructing composite patterns from their constituents, pattern valued functions, and pattern matching operations which determine if a string S is an instance of the pattern P. The pattern matching process may be extremely complex involving the matching of a sequence of subpatterns, back-tracking if a partial match of subpatterns cannot be completed into a complete match, and possible side effects during pattern matching caused by assignments triggered by subpattern matching. The development of Snobol 4 has considerably advanced both our theoretical understanding of the nature of one dimensional (string) patterns and our ability to manipulate such patterns. A good theoretical discussion of Snobol 4 patterns is given by Gimpel [33].

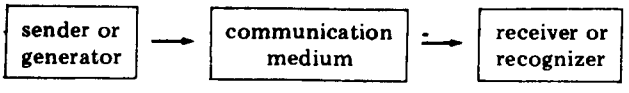
Another nice feature of Snobol 4 is its programmer defined data types facility which allows selector names for each field of a structured data type to be easily defined. The mechanisms for defining, creating and manipulating data types are greatly simplified because no explicit type information need be specified in the program.

The use of the Snobol 4 data definition mechanism in defining and using Lisp data structures will be briefly illustrated. The data type definition "DATA('CONS(CAR,CDR)')" defines a new data type called CONS with two subfields called CAR and CDR. The assignment statement "X = CONS('A', 'NIL')" constructs an initialized instance of this data structure and assigns it to X. The expression "CAR(X)" selects the first subfield "A" of the data structure assigned to X. The naturalness of Snobol 4 for specifying nested construction and selection for programmer defined data structures is illustrated by the assignment statement "Y = CONS('B', CONS('A', 'NIL'))" and the expression "CAR(CDR(Y))" which retrieves the CAR subfield of the CDR subfield of Y (which happens again to be the element "A").

Although Snobol 4 is a relatively rich and complex language its implementation appears to be an order of magnitude simpler than PL/I or Algol 68. In order to increase portability of the language, it has been defined in terms of a relatively machine-independent macro-language, and can be implemented on a new machine simply by implementing the macro-language. An efficient compiler—the Spitbol compiler [18]—makes Snobol 4 competitive for a wide range of nonnumerical programming problems.

M14—Language theory, 1948–1962: Whereas milestones M1–M13 were concerned largely with the development of programming languages, milestones M14–M23 will be concerned with concepts and theories in the programming language field. The topics to be considered include language theory, models of implementation, language definition, program verification, semantics and abstraction. The starting point both historically and conceptually, is the development of language theory.

Both natural languages and programming languages are mechanisms for the communication of messages from a "sender" or "generator" to a "receiver" or "recognizer."



This model of communication was used by Shannon in the late 1940's in developing a mathematical theory of communication [73]. It was used in the late 1950's by linguists and psychologists, such as Chomsky and Miller [16], in the development of a theory of natural languages. In the field of computer science, the great success of the generative (context-free grammar) definition of Algol 60 [63] led to the generative specification of language syntax for all subsequent programming languages, and to the systematic use of recognizers (finite automata and pushdown automata) in implementing translators and interpreters.

The study of natural languages concerns itself with the study of *mental* mechanisms that allow *human* senders and receivers to generate and comprehend a potentially infinite class of sentences after having encountered and learned only a small finite subset of the set of all possible sentences in a language. The study of computer languages is similarly concerned with finite structures that allow languages with an infinite number of sentences to be defined. However, in the case of computer languages, we are not restricted to the study of preexisting human mental mechanisms, but can create language generating and recognition mechanisms with nice mathematical and computational properties. The language generating mechanisms are called *grammars* while the language recognition mechanisms are called *automata*.

One of the most important results in language theory is due to Chomsky, who defined a hierarchy of grammars (type 0, 1, 2, 3 grammars) and a hierarchy of automata (Turing machines, linear bounded automata, pushdown automata, finite automata) and proved the following re-

markable four-part result concerning the equivalence of language generating power of grammars and language recognition power of automata.

1) A language L can be generated by a type 0 (unrestricted) grammar iff it can be recognized by a Turing machine.

2) A language L can be generated by a type 1 (context-sensitive) grammar if it can be recognized by a linear bounded automaton.

3) A language L can be generated by a type 2 (context-free) grammar iff it can be recognized by a pushdown automaton.

4) A language L can be generated by a type 3 (finite-state) grammar iff it can be recognized by a finite automaton.

Proof of the above result provides a number of interesting insights concerning the relation between the processes of language generation and language recognition. Moreover, the four part hierarchy allows us to distinguish between type 0 and type 1 grammars and automata, which are primarily of theoretical interest, and type 2 and type 3 grammars and automata, which are occasionally useful in compiler construction. Much of the practical work in language theory is concerned with the characterization and study of subclasses of type 2 and type 3 grammars and automata.

M15—Compiler technology and theory 1960–1970: The notion of a compiler was developed in the early and mid 1950's by Hopper, the developers of Fortran and many others. By the early 1960's the notion that compiling was a three phase process consisting of lexical analysis, parsing and code generation had been firmly established. During the 1960's there was a great deal of both practical and theoretical work on the mechanization of lexical analysis and parsing [28]. Lexical analysis was modeled by finite automata while parsing was modeled by various subclasses of context-free grammars, such as precedence grammars, $LR(k)$ grammars and $LL(k)$ grammars. The mechanization of code generation proved to be more difficult because it was target language dependent but there was some progress in this area also. The cost of building compilers of given complexity decreased considerably in the 1960's as our understanding of compiler structure increased. In the late 1960's and 1970's there was considerable work on program optimization using techniques such as interval analysis for analyzing the flowchart of a program. The state of the art in compiler technology and theory is ably summarized in [1].

M16—Compiler Compilers: Since there is a lot of similarity between compilers for different languages, the notion was developed of a program which, when primed with the syntatic and semantic specification of a given programming language L, would create a compiler for the programming language L. This concept led to interesting work on specifying the compiler-oriented semantics of programming languages by rules for translating source language constituents into the target language. However, the creation of a working compiler compiler which could actually be used in the production of compilers for new languages or new target machines proved to be too ambitious, because the complexity and diversity of languages and machines is simply too great to permit automation. The purely syntactic task of creating an efficient automatic parser from a BNF syntax specification of a language is possible for certain restricted classes of grammars such as precedence grammars, but becomes unmanageable for more ambitious classes of grammars such as $LR(k)$ grammars because of the difficulty of automatically constructing the tables required for automatic parsing. The automation of compiler semantics is even more difficult than the automation of parsing. The best documented example of a compiler compiler (compiler generator) is probably [58].

M17—Models of implementation, interpreters, 1965–1971: Programming languages such as Fortran and Algol 60 have a lot of "surface complexity" but derive their "integrity" from a simple underlying model of implementation, which specifies how programs are to be executed. A model of implementation is a programming language interpreter rather than a compiler. In the early 1960's compiler models of programming languages were emphasized because compiler construction was a pressing technological problem. By the late 1960's it was realized that interpreter models captured the important characteristics of programming languages much more directly than compiler models, so that serious students of programming language structure discarded compiler models in favor of interpreter models.

Fortran is based on a model in which subroutines and COMMON storage areas occupy fixed size blocks, each object is characterized by a relative address relative to the beginning of its block, and no storage allocation is performed during execution. This model gives rise to language restrictions against arrays with dynamic bounds and recursive subroutines.

Algol 60 is based on an activation record stack model of implementation [84] which can handle recursive procedures and arrays with dynamic bounds but cannot handle pointer-valued variables or procedures which return procedures as their values. Algol 68 can be clearly modeled by a run-time environment with two stacks and one heap [86]. Simula 67 requires each created instance of a class to be modeled by a stack. PL/I has no clean model of implementation, and its lack of integrity may be due precisely to the fact that the language designers were more concerned with the synthesis of source language features than with the development of an underlying model of implementation.

The notion of a model of implementation is important because it pinpoints the simple starting point from which the apparent complexity of a programming language is derived. Man is inherently incapable of handling or manipulating great complexity so it stands to reason that there is some simple internalized model that is used as a starting point for designing complex structures such as programming languages. It is argued here that the simple starting

point for developing a complex programming language may well be a model of implementation in the mind of the designer.

The 1971 conference on data structures in programming languages [26] contained several papers on models of implementation including a paper on the contour model by Johnston [42], a paper on the B 6700 by Organick and Cleary [66], and a paper on data structure models in programming languages by Wegner [87].

Wegner [86] proposed a class of models called information structure models for characterizing models of implementation by their execution time states and state transitions. An information structure model is a triple $M = (I,I^0,F)$ where I is a set of states $I^0 \subseteq I$ is a set of initial states and F is a state transition function which specifies how a state S can be transformed into a new state S' by the execution of an instruction. A computation in an information structure model is a sequence $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots$ where $S_0 \in I^0$ and S_{i+1} is otained from S_i by the execution of an instruction (state transition).

The Lisp APPLY function and the Vienna definition language, discussed in the next section, can be characterized very naturally by information structure models.

Specific assumptions about the structure of the state I and the state transition function F give rise to specific models of implementation. For example states of a Turing machine may be described in terms of three components (t,q,i) where t is the current tape content, q is the current state, and i is the position of the input head. Finite automata are distinguished from Turing machines by the fact that the state transition function F cannot modify the tape component. Pushdown automata have an additional state component called a pushdown tape with characteristic transformation properties.

Programming languages have more complex states and state transitions than automata but may generally be characterized by states with three components (P,C,D) where P is a program component, C is a control component, and D is a data component. Programming languages may be classified in terms of attributes of the P,C,D, components associated with models of implementation. For example the model of implementation of Algol 60 assumes an invariant (reentrant) program component P, a control component C consisting of an instruction pointer ip and an environment pointer ep, and a data component D which is an activation record stack. Fortran does not require P to be reentrant, but requires the size of P,C, and D to be fixed prior to execution.

Information structure models provide a very natural framework for describing programming languages and systems operationally in terms of a specific, possibly abstract, model of implementation. This approach goes against the conventional view that higher level languages should be defined in an implementation-independent way. However, implementation-dependent models reflect the fact that programming-language designers and system programmers think in implementation-dependent ways about programming languages. Implementation-dependent models are therefore valid and important for language designers and system programmers, while implementation independent models are important in other contexts such as program verification.

M18—Language definition, 1960–1970: It is convenient to distinguish between interpreter-oriented language definitions which define the meaning of programs and program constituents in terms of their execution-time effect and compiler-oriented language definitions which define the meaning of source programs in terms of compiled target programs of a target language.

The Algol report is an example of an early (1960) interpreter-oriented language definition with verbal definitions of the meaning of source program constituents. The Lisp APPLY function is an interpreter-oriented language definition in which the execution time effect of source language constructs is rigorously specified by a program.

During the 1960's the interest in compiler technology gave rise to a number of compiler-oriented definitions such as the definition of Euler [89]. Feldman and Gries [28] includes a good review of compiler-oriented language definitions. Knuth [43] proposed an interesting compiler-oriented method of defining the semantics of context-free grammars by associating inherited and synthesized attributes with each vertex of the parse tree of language strings.

During the late 1960's interpreter-oriented definitions of programming languages came back into fashion. The Algol 68 report [80] uses a powerful syntactic notation (VWF notation) to define syntax and semiformal English to define interpreter-oriented semantics. The Vienna definition language [48], [86] is an extension of the Lisp APPLY function definition technique which allows complex languages like PL/I to be defined in terms of an execution-time interpreter.

The Vienna definition language is probably the most practical of the above-mentioned language definition mechanisms. However, it requires approximately 400 pages of "programs" to define PL/I and about 50 pages to define Algol 60. The work on language definition suggests that languages like PL/I are inherently complex in the sense that there simply is no simple way of defining them.

Programming language definitions are intended to serve at least the following two purposes.

1) As a specification of "correctness" for the language implementer.

2) As a specification of "correctness" for the user who wishes to determine whether a program performs its intended task.

The language definition of Algol 60 served as an important frame of reference for a spirited discussion of ambiguities and trouble spots [44]. It was sufficiently precise to serve as an informal tool in checking implementation correctness and program correctness, but was of little help in developing formal methods of program verification. Tools for specifying formal (axiomatic) models of programming languages were developed in the late 1960's and led to an intensive effort in the 1970's to develop

tractable formal language definition models [37], [71].

It is important that programming languages of the future have tractable formal language definitions so that program correctness can be formally determined. One of the objectives of programming language design in the 1970's is "simplicity" where simplicity is increasingly defined in terms of ease of developing a formal definition.

M19—Program correctness, 1963–1969: A program is said to be correct if it correctly performs a designated task (computes a designated function). A program may be thought of as a "how" specification and the designated task or function as an associated "what" specification. A correctness demonstration is a demonstration that the how specification determined by the program is a realization (implementation) of the independently given what specification.

Program correctness was considered by McCarthy (1962) [57], Naur (1965) [64], Dijkstra (1966) [19], Floyd (1967) [29] and Hoare (1969) [36]. Floyd developed the axiomatic approach to program correctness which specifies axioms for primitive program statements and a rule of inference for statement composition. Input-output relations of composite programs may be derived as theorems from input-output relations for primitive statements using the rule of inference for statement composition.

Hoare [36] developed a linear notation for the Floyd formalism. Both axioms and theorems have the form $\{P\}S\{Q\}$ where S is a program statement, P is a precondition, and Q is a post condition. Hoare stated axioms for assignment statements, if-then-else statements and while statements, thus producing a formal system sufficient to prove theorems for programs written in a strict structured programming style. Subsequently, Hoare, together with Wirth, developed a formal definition of Pascal [37] which has been widely used as a starting point for correctness proofs by research workers in program verification.

The axiomatic approach has been widely used for proving the correctness of "small" programs [50], but there are some unresolved problems which prevent its being used as a standard tool for program verification in a production environment. One of the principal limitations of correctness proof techniques is that such techniques are applicable only when the what specification of a program can be given in a simple functional form. The majority of large problems have intractable what specifications (requirements specifications) which may be several hundred pages long, and constantly changing. Thus, it may turn out that formal correctness proofs are simply not applicable to "real" problems, being applicable only to "toy" problems with simple functional what specifications.

M20—Verification, testing and symbolic execution: Program verification may be regarded as an ambitious attempt to prove the correctness of program execution for *all* elements of an infinite input domain and may be contrasted with program testing which is concerned with establishing correctness for individual elements of the input domain. Program correctness for subsets of the input domain may be established by a technique called symbolic execution which is intermediate in generality between program verification and program testing.

The concept of symbolic execution arrived on the scene relatively late and was first publicly presented in 1975 at the international conference on reliable software [41] in papers by King [47] and Boyer, Elspas, and Levitt [6]. It involves the tracing of execution paths of a program with symbolic values of program variables. The set of all execution paths of a program may be thought of as a (possibly infinite) execution tree. Terminal nodes of the tree represent completed execution paths. When a terminal node is reached during symbolic execution then symbolic relations between input and output values for that terminal node are available and program correctness (or incorrectness) can be determined for the subset of values of the input domain which cause the particular execution path to be executed.

Symbolic execution is marginally easier than complete program verification because it is unnecessary to determine loop invariants of program loops. However, other problems which arise in program verification such as the algebraic simplification of algebraic expressions along an execution path are, if anything, more acute because "unfolding" of loops in symbolic execution requires longer sequences of algebraic transformations to be handled. The problem of keeping track of the input domain associated with an execution path is also very difficult. The difficulty of this problem is illustrated by the fact that the "emptyness problem" for execution paths is undecidable. That is, we cannot in general determine whether the input domain associated with a given execution path is empty.

Program testing for a particular value of the input domain is clearly easier than symbolic execution or complete verification since it only involves running the program for the particular input value. However, the key problem in testing is to determine "good" test cases by means of a test data selection criterion.

We may think of a "good" test case as a representative of an equivalence class of "similar" data values with the property that correct execution of the test case increases our confidence in the correctness of the program for all data elements in the equivalence class. If we can partition the input domain into a finite, relatively small, number of such equivalence classes then testing of the program for one element of each equivalence class should increase our level of confidence in the correctness of the complete program.

A number of alternative criteria may be used for determining such equivalence classes. For example the set of all data values associated with a given control path is an example of such an equivalence class. Alternatively, we may directly partition the input domain into input equivalence classes (such as large, medium, and small). Equivalence classes based on the internal program structure which systematically select test cases to exercise all control paths are on the whole more effective than arbitrary equivalence classes imposed on the input domain.

Test data selection criteria can be developed by *program*

structure analysis (control path analysis) *operational profile analysis* (classification of inputs by expected frequency of use) and *error analysis* (testing for specific kinds of errors). The papers by Goodenough and Gerhardt [35], Brown and Lipow [7], and Schneiderwind [74], all presented at the International Conference on Reliable Software [41], illustrate these three approaches to test data selection.

M21—Program verification, program synthesis and semantic definition [98]: Program verification is the process of verifying that a given program Prog correctly performs the task specified by a predicate P. If we are given axioms of the form $\{Q\}S\{P\}$ for a set of primitive statement types and an axiom for statement composition then verification that a program Prog correctly performs the task P requires us to prove the theorem $\{true\}\text{Prog}\{P\}$. That is, the postcondition P for the program Prog implies the precondition *true*.

In the case of program synthesis, we are given a specification of a task P and are required to find a program Prog that correctly performs the task. Program synthesis clearly involves program verification of the synthesized program P as a subtask. However, verification need be performed only for the class of programs which can be synthesized and not for all possible programs of a programming language. Systematic (or automatic) program synthesis avoids unnecessary complexity resulting from bad programming and might actually turn out to be easier than the development of a general purpose verifier for both good and bad programs.

The object of program synthesis is to convert a static description P of what is to be computed into a dynamic description Prog of how it is computed. This can be done in a structured way by the stepwise introduction of dynamic features into the static description. At each step one or more statically defined components is expanded into a structure composed of dynamically defined components which may have inner statically defined components as parameters. A structured development of a program Prog from a specification P consists of a sequence $P_0, P_1, \cdots, P_n$ of successively more dynamic descriptions of P where $P_0 = P, P_n$ = Prog and P_{i+1} is obtained from P_i by "expanding" a component of P_i into a more dynamic form. A formal system such as Hoare [36] may be used to prove that P_{i+1} realizes P if P_i realizes P. Examples of this approach are given by Manna [59], Wirth [90] and Mills [60].

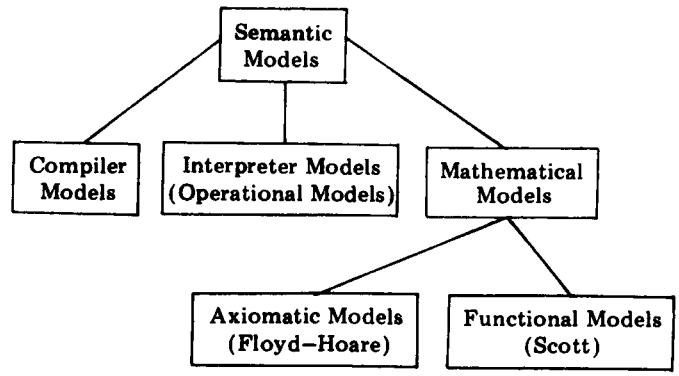
A semantic definition of a programming language L is a mechanism which, given an arbitrary program Prog $\in L$, defines the "meaning" of the program. If the task specification P for a program Prog is taken to be the meaning of Prog, then the semantic definition supplies P given Prog and may be regarded as an inverse process to program synthesis (which supplies Prog given P).

It is very reasonable to think of the input-output predicate P as the meaning of program Prog whenever Prog determines a well defined input-output relation. Unfortunately, there are programs (with an undecidable halting problem) which have no associated input-output predicate P and therefore would have no "meaning" using this notion semantics. Since a semantic definition of a programming language L should associate a meaning with *all* programs of the programming language, this method of assigning meaning is not altogether satisfactory. The set of meanings expressible by input-output predicates P is restricted to the set of recursive functions while the set of meanings expressible by programs is the richer set of recursively enumerable functions.

Floyd [29] called his seminal paper or program verification "Assigning Meaning to Programs," implying that a formal system for program verification also provides a framework for program semantics. It is often convenient for practical purposes to think of the meaning of a program Prog as its input-output predicate. However, input-output semantics determined by axiomatic models is incomplete because the domain of meanings is not sufficiently rich to express the meaning of all programs. In order to achieve completeness, mathematically more sophisticated semantic theories such as those of Scott [71], [80] must be used which map programs into partial recursive functions rather than total recursive functions.

M22—Semantic models: In order to clarify the notion of semantics, it is convenient to introduce the notion of a semantic model as a triple $M = (E,D,\phi)$ where E is a syntactic domain (of programs) D is a semantic domain of denotations and ϕ is a semantic mapping function which maps elements $e \in E$ of the syntactic domain into their denotations $\phi(e) \in D$.

Semantic models for programming languages may be classified in terms of the nature of the domain D of denotations. In particular it is convenient to distinguish between compiler models in which the semantic domain D is a set of programs in a target language, interpreter models in which the meaning of a program is defined in terms of the computations to which it gives rise, and mathematical models in which the meaning of a program is defined in terms of the mathematical function it denotes. Mathematical models may in turn be subdivided into axiomatic models which restrict the semantic domain to total functions and specify functions by a relation between a precondition (inputs) and a post condition (outputs), and functional models (such as those of Scott [71]) in which the meaning of a program is given by an abstract (partial recursive) function. The relation among these models is given by the following figure:



The above discussion makes it clear that the semantics (meaning) of a program is not an absolute (platonic) notion but rather a relative notion which depends on the context of discourse. When we are concerned with compiling, it is natural to think in terms of a compiler oriented semantics for programs. When we are concerned with the process of execution, it is natural to make use of an interpreter oriented semantics. When we are concerned with program verification, then axiomatic semantics is appropriate. When programs are regarded as abstract mathematical objects then the functional semantics of Scott is appropriate.

Each group of semantic models has given rise to a subculture of computer science with its own group of researchers. The subcultures associated with compiler models, interpreter models and axiomatic models have already been discussed (in the sections on compiler methodology, models of implementation and program verification). The Scott approach is the most abstract and Scott's notion of "meaning" has perhaps a greater claim than any other to be considered *the* (platonic) meaning of a program. However, one difficulty with Scott's notion of meaning is that the difference between the how specification of a program and the what specification as an abstract function is so great that the mapping from programs to functions cannot be effectively performed. If it could be effectively performed, then we could decide whether two programs realize the same function by mapping them onto their abstract functions and checking for identity. However, we know that the problem of determining whether two programs realize the same function is undecidable (not even partially decidable) and therefore conclude that the semantic mapping function from programs to abstract functions cannot be constructive.

M23—Abstraction [91]: An abstraction of an object (program) is a characterization of the object by a subset of its attributes. The attribute subset determines an equivalence class objects containing the original object as an element. The objects in the equivalence class are called refinements, realizations or implementations of the abstraction. If the attribute subset captures the "essential" attributes of the object then the user need not be concerned with the object itself but only with the abstract attributes. Moreover, if the attribute subset defining the abstraction is substantially simpler than its realizations then use of the abstraction in place of a realization simplifies the problem addressed by the user.

The input-output relation realized by a program is an example of a program abstraction. It determines an equivalence class of programs (the set of all programs realizing the given input-output relation). Any program in the equivalence class is a realization (refinement) of the abstraction. The input-output relation captures the essential behavior of the program. When the input-output behavior is a simple or well known mathematical function then use of the abstraction in place of a realization serves a useful purpose.

The input-output relation determined by a program may be thought of as a *what* specification (of what the program does) while the program itself is a *how* specification (of how the program is realized). We may, in general, think of an abstraction as a what specification and of its realizations as associated how specifications. The process of abstraction is useful if the what specification characterizing the essential attributes of an object is substantially simpler than the how specification.

Unfortunately, the what specification for programs is not always simpler than the how specification. A program is a relatively compact specification of a functional correspondence between arbitrarily large input and output domains and there is no reason why an explicit description of the input-output relation in a mathematical notation should be simpler than the implicit description by the program. In fact, programs are a more powerful notation for describing functional correspondences than input-output relations because programs can describe recursively enumerable functions (including functions with an undecidable halting problem) while input-output relations can describe only recursive functions (for which the halting problem is decidable).

The equivalence class of all programs (algorithms) associated with a given functional abstraction is studied in the analysis of algorithms. Such equivalence classes can be extraordinarily rich. For example, Knuth in [45] develops an enormous number of different programs for the problem of sorting. It can be shown that the problem of determining whether two programs realize the same abstraction is undecidable (not even partially decidable). The study of the structure of equivalences of how specifications realizing a given what specification is of interest both for programs and other kinds of abstraction.

The notion of abstraction is important in the study of program modularity. All forms of modular programming are concerned with breaking a complex task into modular components where each component has a what specification (abstraction) specifying what the module accomplishes and a how specification (refinement) which specifies how the what specification is realized. If the how specification is specified in terms of a collection of modules which are what specifications to lower level how specifications, then we are led to stepwise abstraction and stepwise refinement. The process of stepwise refinement is illustrated in [88].

The notion of abstraction arises in many different disciplines and may always be characterized in terms of a relation between an equivalence class specification and elements of the equivalence class. The problem of specifying abstractions (equivalence classes) as well as the problem of characterizing the structure of the space of realization (elements) is of interest in many domains of discourse. However, the tools for studying the specification problem and the equivalence problem is determined by the nature of the elements in the domain of discourse. We have already discussed the nature of the specification and equivalence problems when our elements are programs. In the section on "modularity" we will consider the spec-

ification and equivalence problems for a class of modules called *data abstractions* which cannot be completely specified by an input-output relation because they have an internal state.

M24—Pascal [92]: Although Pascal was developed in the late 1960's, its structure and design objectives make it a language of the 1970's. Its designer, Wirth, participated in the early stages of design of Algol 68 as a member of the IFIP working group 2.1, but felt that the generality and attendant complexity of the emerging language was a step in the wrong direction. Pascal, like Algol 68, was designed as a successor to Algol 60. However, whereas Algol 68 aimed at generality, Pascal was concerned with simplicity at the conceptual level, the user level and the implementation level. Conceptual simplicity allows simple axiomatization which facilitates verifiability. User simplicity gives the programmer a better understanding of what he is doing and results in more readable, better structured programs with fewer errors. Simplicity of implementation enhances efficiency and portability and ensures simplicity of the associated operational semantic model.

Pascal provides richer data structures than Algol 60, including records, files, sets and programmer defined type specifications but is otherwise as simple as possible. For example, it excludes arrays with dynamic bounds so as to enhance compile time type checking, and excludes pointers and parameters called by name in the interests of conceptual and user simplicity. The notion of compile time checkable data types is central to the structure of Pascal and provides a degree of program redundancy that enhances program reliability. Control structures are designed so as to encourage good programming style such as that advocated in structured programming.

Because Pascal is conceptually simple, it has been possible to develop a fairly complete formal definition for the language [37]. The existence of this formal definition has in turn led to the widespread use of Pascal as a base language for program verification research [50]. The availability of an axiomatized language has removed one of the obstacles to the development of automatic program verification systems, thus allowing researchers to focus more explicitly on other more formidable obstacles such as the handling of tasks with complex or intractable what specifications.

Pascal and Algol 68 represent two very different approaches to the development of a successor to Algol 60. Although the verdict is not yet in, it may turn out that the Pascal approach will turn out to be more relevant to the development of future programming languages than the Algol 68 approach. However, the discussion of "the APL phenomenon" below indicates that the demands of interactive programming may require us to discard notions such as block structure and explicit type declarations which are fundamental to both Pascal and Algol 68.

M25—The APL phenomenon: The idea of time sharing caught the imagination of the computing community as early as 1960, and led to the development of a number of on-line languages in the early 1960's. Quiktran [60] was developed in 1961–1963 by IBM as an on-line dialect of Fortran but never caught on, perhaps because it could not be adequately supported by existing technology. Joss [75] was developed in 1963–1964 by Shaw and others at the Rand Corporation. Basic (beginners all purpose symbolic instruction code) [48] was developed in 1965–1966 at Dartmouth and has had great success in high schools, two year colleges, and other environments concerned with teaching elementary programming.

APL was developed by Iverson in the early 1960's [40], was implemented as an interactive language in 1967 [30], and has proved to be enormously popular in the 1970's among engineers and mathematicians who need a versatile "desk calculator" to aid them in their work.

APL has a richer set of operators than conventional languages like PL/I or Pascal, including ingeneous extensions of scalar operations to vector and matrix operations which allow loop control structures of conventional programming languages to be implicitly specified in APL. Its emphasis on expressive power at the level of expressions is appropriate to on-line languages, since use of on-line languages in the desk calculator mode is largely concerned with the evaluation of expressions. The richness of APL operators and expressions permits a far greater number of essentially different ways of accomplishing a given computation than in conventional languages. The greater scope for programmer ingenuity leads to greater programmer satisfaction but may lead to programs that are more difficult to read, debug, or maintain.

APL has an explicit mechanism for specifying scopes of identifiers, but has a mechanism for specifying local variables of subroutines. Workspaces are a very effective APL mechanism for defining "modules" containing named subroutines and data sets. There are APL extensions such as APL*PLUS and APL SV [34] specifically designed to allow use of APL for large data processing applications.

APL has no explicitly typed variables or block structure and has the *go to* statement as its only form of transfer control. There is not even an "if-then-else" statement, and conditional branching is performed by an implementation trick (branch to a label 0 is interpreted as exit from a subroutine and branch to an ill-formed label is interpreted as a "continue" statement with no effect). In these respects the structure of APL differs markedly from the current conventional wisdom of the software engineering community. However, it nevertheless strikes a strong responsive chord among practical programmers, indicating that explicit type declarations, block structure and control structure might possibly be discarded in future on-line languages, perhaps because the potential gains in program efficiency and reliability are insufficient to offset the extra program complexity resulting from redundant constituents and additional interrelations among program-constituents.

Arguments *against* block structure, explicit types and explicit control structures may be formulated as follows.

Argument against block structure: One of the original reasons for block structure was the savings in storage re-

sulting from overlays of variables in disjoint blocks. The price paid for this rather trivial saving is an inflexible set of interrelations among program identifiers which adds greatly to the program complexity. APL has scoping mechanisms at the subroutine and workspace level, but none at the block structure level. This looser scoping mechanism appears to be very appealing to practical programmers. Prior to 1970, we might have dismissed the tendency towards looser scoping as being due to a lack of education. However, now that we have become complexity conscious, we can see that block structure imposes additional complexity on a program and that the desire to ruthlessly prune such complexity by eliminating block structure may be justified by the canons of software engineering.

Argument against explicit type declarations: APL is designed so that types of variables may be determined implicitly by context, and there are in fact many syntactic checks on type compatibility between operators and operands in an APL system. Implicit type definitions may well correspond much more closely to the programmers intuitive thought processes than explicit type definitions. Moreover, explicit type declarations greatly increase the number of interactions among program constituents, and therefore increase the complexity of the program. If the programmer needs explicit information about types, APL has query facilities for providing such information to the programmer.

Argument against explicit control structures: The rich operator structure of APL often allows explicit loops and other explicit control specifications to be avoided. Since control structures are probably the single most significant cause of program complexity, languages which allow control structures to be specified implicitly rather than explicitly clearly give rise to textually simpler programs.

Since language usage in the future is likely to become increasingly interactive, and APL is probably the most widely used interactive language, language designers should analyze very carefully the reasons for the popularity of APL. It is not at present clear how much the popularity of APL is due to the quality of its programming system and how much it is due to the quality of the language design. However, it may well turn out that programming languages of the future will be more APL-like than Pascal-like.

M26—Structured programming: The term "structured programming" was introduced by Dijkstra in 1969 in a seminal paper entitled "Notes on structured programming" [24]. These notes are the culmination of several years of personal development, documented by his 1965 paper entitled "Programming Considered as a Human Activity" [20] which emphasizes the importance of programming style and program verification and contains the observation that "the quality of programmers is inversely proportional to the density of go-to statements in their programs," and by his 1968 letter entitled "Go-to Statement Considered Harmful" [21] which sparked a debate concerning the role of the go-to statements in programming that is ably summarized by Knuth [46]. Dijkstra's recent book entitled *A Discipline of Programming* [22] reflects his current thinking on the subject.

Structured programming in its purest (narrow) form is concerned with the development of programs from assignment statements, conditional branching (if-then-else) statements and iteration (while-do) statements by statement composition. These statement forms can be nicely axiomatized [36] and correspond to "natural" forms of mathematical reasoning (the if-then-else statement corresponds to enumerative (case analysis) reasoning and the while-do statement corresponds to inductive reasoning). It was shown by Bohm and Jacopini [4] that these statement forms are sufficient for expressing any computable function. Moreover, it turns out that these statement forms are appropriate for many practical problems although they must be supplemented by other statement forms in certain cases such as unusual exit from a loop.

Structured programming in its more general meaning is concerned with the better organization of the program development process to achieve objectives such as simplicity, understandability, verifiability, modifiability, maintainability, etc. In order to achieve these objectives it is important to develop a methodology for the modular decomposition of programs into components suitable both for bottom-up and top-down program development. In this connection, it is convenient to distinguish between "programming in the small" concerned with modularity and program structuring at the primitive statement level and "programming in the large" concerned with modularity at a higher (subprogram and data structure) level. The if-then-else and while-do constructs are appropriate module building constructs for programming in the small. The Algol procedure, Simula class and APL workspace are examples of module building constructs for programming in the large. Current research on modularity will be discussed in a separate section.

Structured programming has affected programming language usage in placing greater emphasis on if-then-else and while-do constructs and deemphasizing the go-to statement. It is likely to affect the design of future programming languages by introducing new kinds of program modules for programming in the large, and by placing greater emphasis on verifiability as a programming language design objective. The availability of appropriate concepts of modularity should help the user in systematic modular program development for complex problems. However, there are important areas of program development, such as choice of an appropriate modular data structure where available tools are of little help to the programmer. The influence of the choice of data structure on program structure is discussed in a paper on "top-down program development" by Wirth [90]. The duality between program structure and data structure is discussed in a provocative way by Hoare [38], [39].

The techniques of structured programming have had an impact not only on academic computer science but also on production programming [8]. The chief programmer team approach developed by Mills and Baker [61] is an example

of a management structure which makes use of structured programming. The New York Times project [9] is perhaps the most widely advertised success story for the chief programmer team approach, claiming a productivity of 10 000 instructions per man year with only one error per man year. However, the reported success of this project was subsequently challenged, on the basis that maintenance and modifiability of the completed program was unsatisfactory. It appears that the chief programmer team approach is designed to optimize program development but pays insufficient attention to the operations and maintenance part of the life cycle (see the section on life cycle).

M27—Structured model building: Specifications of programming languages are effectively complex programs in some specification language. The notions of abstraction structuring, and stepwise refinement are just as applicable to the construction of semantic models (definitions) of programming languages as they are to the construction of applications programs. Thus, the abstract notion of a semantic model (for a specific language) can be realized by a compiler model, interpreter model, axiomatic model, or functional model (see M22). Once the desired class of models has been chosen, there is enormous scope for "structuring" the language definition by first making "high-level" decisions concerning the overall structure of the model and then filling in lower level details by a process of stepwise refinement. The term "partial model" may be used to describe an intermediate partial language specification in this process of stepwise refinement.

The above structured model building approach will be briefly illustrated by showing how stepwise refinement may be used to build an information structure model (interpreter model) of Algol 60 [97]. In the case of information structure models (I,I°,F) the partial models of the stepwise refinement process will have partial (successively more complete) specifications of the state components I,I°, and the state transition function F. The initial model M_0 would be an arbitrary model with no restriction on I,I°,F. A "first-order" model M, might require states I to be of the form (P,C,D) where P is an invariant (read only) program component, C has the form (ip,ep) where ip is an instruction pointer into P and ep is an environment pointer into D, and D is a stack of activation records. A "second-order" model M_2 might then be introduced which defines the state transitions (instructions) for block entry and exit and procedure call and return. Eventually, a final model M_n would completely define the state structure I and state transitions F for every Algol statement.

The partial models which arise in the above stepwise refinement process specify partial (operational) semantics for partial syntax specifications and may be thought of as defining language classes which are abstractions of the language that is being defined. For example, the abstraction "Algol-like languages" may in principle be defined by a partial information structure model which fixes those semantic and syntactic features that are essential if the language is to be Algol-like and leaves open optional language features of Algol-like languages.

The use of structured techniques of model specification is likely to lead to more understandable definitions of a number of existing programming languages. However, an even more potent way of developing programming languages with simple specifications is to use simplicity of specification as one of the criteria of programming language design, as was done in the case of the programming language Pascal.

M28—The life cycle concept: The software life cycle as formalized by department of defense agencies consists of a *concept formulation* and *requirements specification* stage, a *software development* stage, and an *operations* and *maintenance* stage. These stages may in turn be refined so that the software development stage might consist of a requirements analysis stage, a program design stage, an implementation and debugging stage and a testing and evaluation stage. In analyzing three large military software projects, it was estimated that such systems typically have a life cycle of 16 years, consisting of a concept formulation and requirements stage of 6 years, a software development stage of 2 years, and an operations and maintenance stage of 8 years [70]. Thus, the software development stage comprises only one eighth of the total life cycle of a typical large military software project.

The life cycle concept provides a basis for a more complete analysis of software systems than was previously possible. In the 1960's and early 1970's programming projects were organized to minimize software development costs rather than total life cycle costs. This led to a disproportionate emphasis on program design and implementation and a comparative neglect of both the initial determination of what it is that we really want to accomplish and the long years of program usage in an environment which may involve frequent program modification.

Emphasis on the life cycle as opposed to the software development phase affects both programming language design and programming language usage. For example emphasis on software development requires programming languages to be designed for rapid and correct program development while emphasis on the life cycle requires programs to be readable and modifiable during the long operations and maintenance period, providing a strong argument for simplicity of language design. Language usage should be modular, so that modifications of one part of the program do not have unexpected side effects in another part of the program. Clever tricks which make the program less readable should be avoided like the plague.

The life cycle approach allows the systematic study of cost and effort in all stages of existence of a software system [69]. Bottlenecks can be uncovered in the manner of critical path analysis and tools and techniques may be developed for eliminating such bottlenecks. Studies of software systems have in fact uncovered some quite unexpected facts about system behavior such as the fact that 64 percent of errors are system design errors while only 36 percent are system implementation errors. This suggests that design, rather than implementation, is the

bottleneck in software system development and has implications concerning the allocation of funds for research in software engineering.

M29—Modularity: The subroutine mechanism for realizing program modularity was developed as early as 1951 [83]. It was a fundamental feature of Fortran, whose design provided an enormous impetus towards modular programming. Algol 60 was in some ways a backward step from the viewpoint of modularity because its nested module structure discouraged independent module development and because procedure modules could not adequately handle data which remained in existence between instances of execution of a procedure.

The Simula class is a very flexible generalization of the Algol 60 procedure module. It separates creation and deletion of instances of a class from entry and exit for purposes of execution. Coroutine control allows the program and data state at arbitrary points of execution to be preserved and subsequently restored. Access to objects declared in the outermost block of a class provides a more flexible (too flexible) mechanism for module intercommunication. The subclass facility is an ingenious syntactic mechanism for providing the advantages of hierarchical (nested) modular environments while avoiding the need for physical textual nesting of the associated modules. The class concept has served as an inspiration to designers of modular programming languages but is probably too rich in properties to serve as a prototype for modular design.

The collection of declarations in the outer block of a module may be regarded as a set of attributes or resources. One of the purposes of a module is to erect a "fence" around this set of attributes which allows systematic *information hiding* [68] of internal (hidden) attributes of a module and selective specification of a subset of externally known (*exportable*) attributes. Recent research on Clu [51] and Alphard [94] has been concerned with mechanisms for hiding and exporting module attributes.

The experimental language Clu [51] requires its program modules to consist of a collection of exportable procedures operating on a hidden (internal) data structure, and refers to such modules as *clusters*. Clusters are convenient for defining data types (such as stacks) by means of operations (such as push, pop, top, create, testempty) independently of the internal data structure (linear list or array) used to realize the cluster. The user sees an abstraction (the stack abstraction) which is defined by hiding the internal data structure and exporting only the operations. Such an abstraction is called a *data abstraction* because it abstracts from a specific data representation. The effect of the operators is defined by axioms such as "top(push(x,stack)) = x" which defines the effect of the "top" operation in terms of previously executed "push" operations without making any commitment to data representation. Any data structure which causes the defining axioms for the operations to be satisfied is an adequate realization of the data abstraction. Of course, the development of complete and sound sets of axioms for characterizing the set of cluster operations in a data independent way may, in general, be difficult. However, in most practical cases, we can characterize the behavior of "output" operations of a data abstraction reasonably simply in terms of the effect of previous input operations, and a complete set of axioms can be developed by systematically using our knowledge of what the operations are supposed to accomplish.

The experimental modular programming language Alphard [94] calls its modules *forms*. Recent research on Alphard has emphasized verifiability as an objective of the language. Forms have a *representation* component which defines the representation of hidden data structures, an *implementation* component which defines the implementation of external attributes (operators) of the form and a *specification* component which specifies the "abstract" properties of attributes so that the correctness of their implementation can be verified.

Other work on modular programming languages includes Brinch Hansen's development of *monitors* in concurrent Pascal [9] and Wirth's introduction of *modules* in a Pascal-like language for modular multiprogramming called Modula [97]. Both monitors and modules are motivated by the need to provide the user with machine independent abstractions of machine resources such as disks, user consoles, synchronization primitives, etc. In [9] the relation between the implementation and user abstraction for monitors is described by considering how monitors are implemented. In [97] the relation between abstraction and implementation of modules is described at the language level by introducing notions such as *define list* of objects defined in the module for use outside the module and a *use list* of objects declared outside the module and used inside the module.

Among the problems which must be addressed in any modular programming language are the problems of module interface definition and module interconnection. These problems had already been identified in the 1950's in connection with the development of Fortran (the transfer vector mechanism). One recent example of work in this area is the thesis by Thomas [79] who develops a module interconnection language (MIL) for specifying module interfaces in terms of inherited attributes (use lists), synthesized attributes (define lists), and locally generated attributes using a model similar to Knuth's attribute grammars [43].

Although recent research on modularity and abstraction has greatly increased our understanding of how modules may be designed, it is not yet clear how effectively these notions can be incorporated in future programming languages. The explicit support of clusters or Simula classes introduces extra complexity into a language both at the level of verifiability and at the level of implementation, determining complete and consistent specifications for clusters to serve as a starting point for formal verification. Simula 67 has not become a widely used application language in spite of its superior modularity facilities. The modular programming facilities of a language are clearly among its most important design features, and it is quite likely that future programming languages will contain new

kinds of primitives for defining both program and data abstractions. But the precise nature of these primitives has not yet been determined.

M30—Data oriented programming languages: We may distinguish between the program-centered and data-centered views of programming. The program-centered view emphasizes program development and considers data only piecemeal as and when it becomes the object of program transformation. This view is appropriate to numerical problems involving complex functional transformations on simple data structures. The data-centered point of view considers the data structure (data base) as the central part of a problem specification and views programs as "bugs" which crawl around the data base and occasionally query, update or augment the portion of the data base at which they currently reside. This view is appropriate for airline reservation systems, management information systems, information retrieval systems or any other systems whose state description requires a complex data structure and whose operations (transactions) are local queries or perturbations of that data structure.

Bachman in his Turing lecture [11] compares the shift from the program-centered to the data-centered point of view with the shift from an earth-centered to a sun-centered model of the universe brought about by the Copernican revolution. There is no doubt that the increasing importance of the data-centered view of programming will affect the design of future programming languages.

Since programming was initially motivated by numerical problems early programming languages and programming methodology emphasized the programming centered point of view. Algol 60 blocks and procedures are examples of program-centered constructs since internal data structures are forced to disappear between instances of execution. Simula classes generalize block structure so that it becomes appropriate for data-centered programming.

Cobol is an example of an early programming language which allows program and data to be handled in a symmetrical fashion. Programming systems for data-centered programming developed during the 1960's include IDS (integrated data store) [5], and IMS (information management system) [25]. The data-centered view of programming led in the 1970's to the development of data-base languages and systems [14], [25].

Data-base systems may be classified [25] into *network systems* which require the user to view the data base as a network (spagghetti bowl); *hierarchical systems* which require the data base to be tree structured, and relational data-base systems which permit the user to view the data base as a set of abstract relations. Network and hierarchical systems give rise to "low level" data base languages since they require the user to be explicitly aware of the data structure implementation. Relational systems give rise to high level data base languages which allow programs to specify transactions independently of the internal data structure representation, but lead to formidable implementation problems.

Network systems are a direct outgrowth of the work of the Codasyl data-base task group (DBTG) [14] and were heavily influenced by Bachman's work on IDS [5]. Hierarchical systems are the simplest class of data-base management systems, and most of the practical systems of the 1960's such as IMS were hierarchical. The relational approach to data-base management systems was pioneered by Codd in 1970 [13]. A good recent survey of the state of the art may be found in [77].

There is a great deal of current work on the design and implementation of data-base language. Recently developed relational data-base languages include Sequel [14], Quel [2], and query by example [100].

Conclusions

The above collection of concepts and milestones is by no means complete, but illustrates the great variety of programming language concepts and products developed during the last 25 years. One of the more interesting facts that emerges from a study of programming language development is the remarkable stability of early programming languages like Fortran and Cobol, and the comparative lack of success of subsequently developed languages like PL/I, Algol 68, Simula 67, and Pascal in capturing significant numbers of adherents in a nonuniversity environment. The exception is perhaps APL which has captured the hearts of a new class of user (the desk calculator user).

All the programming languages described in this paper were developed and implemented in the 1950's and 1960's. Although there have been a number of proposals for general purpose languages in the 1970's such as CS 4 [15] and the Tinman requirements specification for a new DOD-sponsored common higher order language [31], no new general purpose languages comparable to PL/I, Algol 68, or Pascal have been launched during this period. The 1970's have been a period of retrenchment in the development of general purpose languages. A number of new insights have been developed such as the importance of simplicity, readability, verifiability, and maintainability in program design and language design. A better appreciation of the concept of modularity has been developed and we have made some gains in our understanding of program verification. But these insights have led to changes in the mode of use of existing programming languages rather than in the design of a new class of programming languages which are so clearly superior that they are automatically accepted as a replacement for existing programming languages.

The demonstrated reluctance of the programming community to accept a new language is due partly to the costs of a changeover, and partly to the natural resistance to changes in technology. It is due partly to the fact that programming language designers have not been able to come up with an acceptable compromise between simplicity and versatility that is a substantial improvement over Fortran or Cobol. However, a further reason may be that programmer productivity is not as sensitive to lan-

guage changes as programming language professionals would like to think. Fortran-like languages provided a significant increment of productivity over assembly language but it may well be that further language refinements cause only marginal or even negative increments in programmer productivity. Programming style, structured programming and other methodologies are largely language independent and are probably far more important in increasing programmer productivity than the development of new languages. Ultimately, it is the quality of programming rather than the programming language that determines the cost and reliability of production programs.

The field of programming languages was central to the development of computer science in the 1950's and 1960's, leading to important practical products and to important theoretical advances in our understanding of the nature of computer sciences. It may well be that programming language professionals did their work so well in the 1950's and 1960's that most of the important concepts have already been developed. The programming language field may play a less central (though still important) role in computer science in the 1970's and 1980's than it did in the 1950's and 1960's.

References

Note: OSIPL refers to [25]. ICRS refers to [40].

[1] A. V. Aho and J. R. Ullman, *The Theory of Parsing, Translation and Compiling*. Englewood Cliffs, NJ: Prentice-Hall, vol. I, 1972; vol. II, 1973.

[2] E. Allman, M. Stonebraker, and G. Held, "Embedding a relational sublanguage in a general purpose programming language"' *SIGPLAN Notices*, Mar. 1976.

[3] M. M. Astrahan and D. Chamberlin, "Implementation of a structured English query language," *Commun. Ass. Comput. Mach.*, Oct. 1975.

[4] C. Bohm and G. Jacopini, "Flow diagrams, turing machines, and languages with only two formation rules," *Commun. Ass. Comput. Mach.*, May 1966.

[5] C. W. Bachman, "A general purpose system for random access, memories," in *FJCC Proc.*, 1964.

[6] R. S. Boyer, B. Elspas, and K. N. Levitt, "A formal system for testing and debugging programs by symbolic execution," *ICRS*, Apr. 1975.

[7] J. R. Brown and M. Lipow, "Testing for software reliability," *ICRS*, Apr. 1975.

[8] F. T. Baker, "Structured programming in a production programming environment," *ICRS*, Apr. 1975.

[9] P. Brinch Hansen, "The purpose of concurrent PASCAL," *ICRS*, Apr. 1975.

[10] F. T. Baker and H. D. Mills, "Chief programmer teams," *Datamation*, 1973.

[11] C. W. Bachman, "The programmer as navigator," (1973 Turing lecture), *Commun. Ass. Comput. Mach.*, Nov. 1973.

[12] *COBOL 1961: Revised Specifications for a Common Business Oriented Programming Language*, U. S. Govt. Printing Office, 1961.

[13] E. F. Codd, "A relational submodel for large shared data banks," *Commun. Ass. Comput. Mach.*, June 1970.

[14] "CODASYL," Data Base Task Group Rep., Apr. 1971.

[15] *CS-4 Language Reference Manual and Operating System Interface*, Intermetrics Publ., Oct. 1975.

[16] N. Chomsky and G. A. Miller, *Introduction to the Formal Analysis of Natural Languages, Handbook of Mathematical Psychology*, vol. II. New York: Wiley, 1963.

[17] D. Dahl and C. A. R. Hoare, *Hierarchical Program Structures, in Dahl, Dijkstra and Hoare, Structured Programming*. New York: Academic, 1972.

[18] R. Dewar, "SPITBOL 2.0," Illinois Inst. Technol. Rep., 1971.

[19] E. W. Dijkstra, "A constructive approach to the problem of program correctness, *BIT*, Aug. 1968.

[20] ——, "Programming as a human activity," *Proc. IFIP Congress*, 1965.

[21] ——, "Go to statement considered harmful," *Commun. Ass. Comput. Mach.* (Lett.), Mar. 1968.

[22] ——, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[23] ——, "Making a translator for ALGOL 60," *APIC Bull.*, vol. 7, 1961.

[24] ——, *Notes on Structured Programming, in Dahl, Dijkstra and Hoare, Structured Programming*. New York: Academic, 1972.

[25] C. J. Date, *An Introduction to Data Base Systems*. New York: Addison-Wesley, 1975.

[26] *Data Structures in Programming Languages, Proc. of Symp., SIGPLAN Notices*, Feb. 1971.

[27] "FORTRAN vs. basic FORTRAN," *Commun. Ass. Comput. Mach.*, Oct. 1964.

[28] J. Feldman and D. Gries, "Translator writing systems," *Commun. Ass. Comput. Mach.*, Nov. 1968.

[29] R. W. Floyd, *Assigning Meanings to Programs, Proc. Symp. App. Math.* vol XIX, AMS, 1967.

[30] A. D. Falkoff and K. E. Iverson, *The APL Terminal System, in Klerer and Reinfelds, Interactive Systems for Experimental Applied Mathematics*. New York: Academic, 1968.

[31] D. A. Fischer, "A common programming language for the department of defense, background and technical requirements," IDA Sci. Technol. Division, paper P-1191, June 1976.

[32] R. Griswold, J. Poage, and I. Polonsky, *The SNOBOL 4 Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1971.

[33] J. Gimpel, "A theory of discrete patterns and their implementation in SNOBOL 4," *Commun. Ass. Comput. Mach.*, Feb. 1973.

[34] L. Gilman and A. J. Rose, *APL, an Interactive Approach*, 2nd Ed. New York: Wiley, 1974.

[35] J. B. Goodenough and S. L. Gerhard, "Towards a theory of test data selection," *ICRS*, Apr. 1975.

[36] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, Oct. 1969.

[37] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," *Acta Inform.*, vol. 2, no. 4, 1973.

[38] ——, *Notes on Data Structuring, In Dahl, Dijkstra and Hoare, Structured Programming*. New York: Academic, 1972.

[39] ——, "Data reliability," *ICRS*, Apr. 1975.

[40] K. E. Iverson, *A Programming Language*. New York: Wiley, 1962.

[41] *Proc. Int. Conf. Reliable Software*, Apr. 1975; also *SIGPLAN Notices*, June 1975.

[42] J. Johnston, "The contour model of block structured processes," *DSIPL*, Feb. 1971.

[43] D. E. Knuth, "The Semantics of Context Free Languages," in *Mathematical Systems Theory*, vol. II, no. 2, 1968.

[44] ——, "The remaining trouble spots in ALGOL 60," *Commun. Ass. Comput. Mach.*, Oct. 1967.

[45] ——, *The Art of Computer Programming Volume III, Sorting and Searching*, 1973.

[46] ——, "Structured programming with go to statements," *Comput. Surveys*, Dec. 1974.

[47] J. C. King, "Symbolic execution and program testing," *Commun. Ass. Comput. Mach.*, July 1976.

[48] J. G. Kemeny and T. E. Kurtz, *Basic Programming*. New York: Wiley, 1967.

[49] P. Lucas and K. Walk, "On the formal description of PL/I," *Annu. Rev. Automatic Programming*, vol. 6, pt 3. New York: Pergamon, 1969.

[50] R. L. London, "A view of program verification," *ICRS*, Apr. 1975.

[51] B. H. Liskov, "A note on CLU," Computation Structures Group Memo 112, Nov. 1974.

[52] B. M. Leavenworth, "Syntax macros and extended translation," *Commun. Ass. Comput. Mach.*, Nov. 1966.

[53] B. H. Liskov and S. N. Zillies, "Specification techniques for data abstractions," *ICRS*, Apr. 1975.

[54] M. D. McIlroy, "Macro instruction extensions to compiler languages," *Commun. Ass. Comput. Mach.*, Apr. 1960.

[55] C. N. Mooers, "TRAC-A procedure-describing language for a reactive typewriter," *Commun. Ass. Comput. Mach.*, Mar. 1976.

[56] J. McCarthy *et al., LISP 1.5 Programmers Manual.* Cambridge, MA: MIT Press, 1965.
[57] J. McCarthy, "Towards a mathematical science of computation," in *Proc. IFIP Congr.*, 1962.
[58] W. M. McKeeman, J. H. Horning, and D. B. Wortman, *A Compiler Generator.* Englewood Cliffs, NJ: Prentice-Hall, 1970.
[59] Z. Manna, *Mathematical Theory of Computation.* New York: McGraw-Hill, 1974.
[60] H. D. Mills, "Mathematical foundations for structured programming," IBM Corp., Gaithersburg, MD, FSC 72-6012, 1972.
[61] J. H. Morissey, "The QUIKTRAN system," *Datamation*, Feb. 1964.
[62] P. Naur, Ed., "Report on the algorithmic language ALGOL 60," *Commun. Ass. Comput. Mach.*, May 1960.
[63] ——, "Revised report on the algorithmic language ALGOL 60," *Commun. Ass. Comput. Mach.*, Jan. 1963.
[64] ——, Proofs of Algorithms by General Snapshots, BIT 6, 1966.
[65] Newell *et al., Information Processing Language V Manual*, 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall, 1965.
[66] E. I. Organick and J. G. Cleary, "A data structure model of the B6500 computer system," *DSIPL*, Feb. 1971.
[67] *PL/I, Current IBM System 360 Reference Manual*, (or Bates and Douglas), 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall, 1975.
[68] D. I. Parnas, "A technique for software module specification with examples," *Commun. Ass. Comput. Mach.*, May 1972.
[69] B. Randell and L. J. Russell, *ALGOL 60 Implementation.* New York: Academic, 1964.
[70] D. J. Reifer, "Automated aids for reliable software," *ICRS*, Apr. 1975.
[71] D. Scott and S. Strachey, "Towards a mathematical semantics for computer languages," PRG 6, Oxford Univ. Comput. Lab., 1971.
[72] J. Sammet, *Programming Languages, History and Fundamentals.* Englewood Cliffs, NJ: Prentice-Hall, 1969.
[73] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communications.* Urbana, IL: Univ. Illinois Press, 1962.
[74] N. F. Schneiderwind, "Analysis of error processes in computer software," *ICRS*, 1975.
[75] C. J. Shaw, "JOSS, a designers view of an experimental on-line system," in *Proc. FJCC*, 1964.
[76] ——, "A specification of JOVIAL," *Commun. Ass. Comput. Mach.*, Dec. 1963.
[77] E. H. Sibley, Ed., "Special issue: Data base management systems," *Comput. Surveys*, Mar. 1976.
[78] A. M. Turing, "On computable numbers with an application to the entscheidungsproblem," in *Proc. London Math. Soc.*, 1936.
[79] J. Thomas, "Module interconnection in programming systems supporting abstractions," Ph.D. dissertation, Brown Univ., Providence, RI, May 1976.
[80] R. D. Tennent, "The denotational semantics of programming languages," *Commun. Ass. Comput. Mach.*, Aug. 1976.
[81] J. Von Neumann, "The EDVAC report," in *Computer from PASCAL to Von Neumann*, H. Goldstein, Ed. Princeton, NJ: Princeton Univ. Press, 1972, Ch. 7, discussion.
[82] V. Wingaarden *et al.*, "Report on the algorithmic language ALGOL 68," *Numer. Math.*, Feb. 1969; also revised report, *Numer. Math.*, Feb. 1975.
[83] M. V. Wilkes, D. J. Wheeler, and S. Gill, *The Preparation of Programs for a Digital Computer.* New York: Addison-Wesley, 1951 (revised Ed., 1957).
[84] P. Wegner, *Programming Languages, Information Structures and Machine Organization.* New York: McGraw-Hill, 1968.
[85] W. Waite, "A language independent macro processor," *Commun. Ass. Comput. Mach.*, July 1967.
[86] P. Wegner, "Three computer cultures, computer technology, computer mathematics and computer science," in *Advances in Computers*, vol. 10. New York: Academic, 1972.
[87] ——, "Data structure models in programming languages," *DSIPL*, Feb. 1971.
[88] ——, "The Vienna definition language," *Comput. Surveys*, Mar. 1972.
[89] N. Wirth and H. Weber, "Euler—A generalization of ALGOL and its formal definition," *Commun. Ass. Comput. Mach.*, Jan. and Feb. 1966.
[90] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, Apr. 1971.
[91] P. Wegner, "Abstraction—A tool in the management of complexity," in *Proc. 4th Texas Symp. Comput.*, Nov. 1975.
[92] N. Wirth, "The programming language PASCAL," *Acta Inform.*, 1971.
[93] P. Wegner, "Structured model building," Brown Univ., Providence, RI, Rep., 1974.
[94] W. Wulf, R. L. London, and M. Shaw, "Abstraction and verification in ALPHARD, introduction to language and methodology," Carnegie-Mellon Univ., Dep. Comput. Sci. Rep., June 1976.
[95] P. Wegner, "Operational semantics of programming languages," in *Proc. Symp. Proving Assertations about Programs*, Jan. 1972.
[96] ——, "Research paradigms in computer science," in *Proc. 2nd Int. Conf. Reliable Software*, Nov. 1976.
[97] N. Wirth, "Modula: A language for modular multiprogramming," ETH Institute for Informatics, TR18, Mar. 1976.
[98] P. Wegner, "Structured programming, program synthesis and semantic definition," Brown Univ. Rep., Providence, RI, 1972.
[99] V. Yngve, "COMIT as an IR language," *Commun. Ass. Comput. Mach.*, Jan. 1962.
[100] M. Zloof, "Query by example," in *Proc. Nat. Comput. Conf.*, 1975.

Peter Wegner received the B.Sc. degree in mathematics from the Imperial College, London, England, the Diploma in numerical analysis and automatic computing from Cambridge University, Cambridge, England, the M.A. degree in economics from Penn State University, and the Ph.D. degree in computer science from London University, London, England.

He has taught at the London School of Economics, Penn State, Cornell University, and Brown University and has been on the staff of the Computation Center at the Massachusetts Institute of Technology, and Harvard University. He is currently with the Division of Applied Mathematics, Brown University, Providence, RI. His publications are primarily in the programming language area but include papers in operations research and statistics.

Dr. Wegner has been consultant to the ACM Curriculum Committee (1965–1968), SIGPLAN Chairman (1969–1971) and is presently a member of the ACM Council.