Procedural Programming Languages for the Development of CAD and CAE System Software

Neal M. Holtz[1] and William J. Rasdorf[2]

Abstract

This paper addresses the need for engineers to have a working knowledge of the fundamentals of computer programming languages. In pursuit of this, it briefly looks at the history behind four of the more well known programming languages. It then attempts to identify, and to look critically at, the attributes of programming languages that significantly affect the production of engineering software. The four "traditional" procedural programming languages chosen for review are those intended for scientific and general purpose programming — FORTRAN–77, C, Pascal, and Modula–2. These languages are compared and some general observations are made.

As it is felt important that professional engineers should be able to make rational decisions about the choice of a programming language, the emphasis throughout this paper is on a methodology of evaluation. Choosing an appropriate language can be a complex task and many factors must be considered; consequently, fundamentals are stressed.

1. Introduction

The complexity of engineering software has increased dramatically in the past decade. In the early years, most engineering applications were concerned solely with solving difficult numerical problems, and little attention was paid to man–machine interaction, data management nor to integrated software systems. Now computers solve much wider variety of problems, particularly those in which numerical computations are less predominant. In addition, completely new areas of applications have recently emerged, such as those addressed by the so–called "expert systems." With the increase in the variety, functionality and complexity of engineering software, with its more widespread use, and with its increasing importance, more attention must be paid to programming language suitability so that rational decisions regarding language selection may be made.

It is important that professional engineers be aware of the issues addressed in this paper, for it is they who must design, acquire, and use applications software, as well as occasionally developing or managing its development [15]. It is becoming virtually impossible for an engineer to perform his duties without the use of large amounts of software; the quality of that software can markedly affect the quality and cost of his work.

[1] Associate Professor of Civil Engineering, Carleton University, Ottawa, Ontario, Canada K1S 5B6.

[2] Associate Professor of Civil Engineering and Computer Science, North Carolina State University, Box 7908, Raleigh, NC, USA 27695–7908.

1.1 Objectives

The objectives of this paper are several:

- to identify the general attributes of programming languages that significantly affect the development and organization of applications software for engineering.
- to contrast and compare four "traditional" programming languages with respect to these attributes.
- to summarize and present this comparison in a form that allows software developers to make a more rational choice of programming language.
- to point out features of programming languages that are evolving or have more recently been developed.

The four languages chosen are *FORTRAN-77*, referred to simply as "FORTRAN" below (because of its pervasiveness, historical importance, and role as an engineering standard), *C* (because of its widespread use in other areas), *Pascal* (because of its simplicity, well formedness, popularity as a teaching language, and wide availability) and *Modula-2* (because of the expectation that it will become a replacement for Pascal). The intent of this paper is not to make general recommendations about a particular language. Rather, the hope is to provide a sound basis for allowing engineers to make rational choices among programming languages. This is done by enumerating and discussing the important attributes of programming languages, and demonstrating the evaluation task on the four languages chosen.

1.2 Historical Review

1.2.1 Evolution

Language origins were very basic. Data, control and organization constructs were initially very primitive. These have evolved to the point where much more sophisticated and powerful constructs exist. Data, for example, originally resided only in variables and homogeneous arrays but has evolved to the point where the programmer can readily deal with a variety of conceptual data structures, including trees, networks, lists, stacks, etc. The implementation of the concept of data abstraction has occurred. Control forms have evolved from simple sequential execution, branching and iteration to more sophisticated forms of each, as well as to other concepts such as recursion, data driven and goal driven programming. Program organization has evolved from totally unstructured collections of lines of code into modular components that are extensively tested, highly specialized and portable software engineering components.

1.2.2 Language Design Criteria

The design of a programming language is of course influenced greatly by its expected use, just as its ultimate usefulness is affected by the care and thought given to its design. The four languages presented here had quite different intended uses, and thus had markedly different design criteria. Any complete review should at least mention these criteria so that developers have some understanding of the differences among programming languages.

FORTRAN, in fact, was not really designed at all. Its intended use was the evaluation of scientific formulae and its developers established efficiency of execution as the most important criteria. In fact, one of them even stated that at that time he did not believe language design itself was difficult, and so relatively little effort and consideration was given to design [3]. That approach resulted in what is now seen to be one of the major disadvantageous of FORTRAN; new concepts have not been and cannot be easily introduced because of the ad–hoc way in which the base was developed.

C was designed to be a general purpose programming language, with an emphasis on systems programming; this means that it had to have good support for data structures, character manipulation and interfacing to the operating system. First implementations were for PDP–11 computers. Certain characteristics of that machine had influence on the design of the language, for example, the inclusion of increment and decrement operators. However, that influence was not excessive and C has moved well to other architectures. Terseness of expression also seemed to be an important design criteria, as the language often sacrifices clarity for brevity.

Pascal was intended primarily as a teaching tool — to teach the fundamentals of computing science using a language that was as concise and succinct as possible. A secondary goal was to combine in one language all those features and ideas recognized to be important ingredients of a language in 1970. Because the emphasis was on a teaching language, some key facilities required for developing large programs, particularly those dealing with separate compilation, were deemed unnecessary. Probably more thought was given to the implementation of data structures than any other area, with the result being that data structuring is recognized as a Pascal strong point.

Modula–2 is an attempt to build on Pascal's strengths (data structuring, conciseness, simplicity, etc.), to alleviate its recognized weaknesses, and to add those features necessary for constructing large programs. A few new programming methodologies, such as partial support for data abstraction, were incorporated. An attempt was made to have the syntax of Modula–2 compatible with that of Pascal wherever possible, though there are some differences.

2. Language Attributes

There are many attributes of programming languages that affect the costs of development, maintenance, and execution of applications programs. This section discusses the more important of these in general terms. The following section summarizes the four programming languages with respect to these attributes.

The fundamental components of *any* computer program are data, control and organization. As these are fundamental to all, they act as a benchmark against which languages may be compared. Given descriptions of these fundamentals, one can judge the comprehensiveness and robustness of a language. If a language is missing one or more of the desired features, one can also judge how easy or difficult it is to simulate those missing features.

Most of the terminology and concepts in this section are taken from Wulf [19], which is an excellent reference on the fundamentals of computer programming. Other notable references are Elson [7], and MacLennan [13].

2.1 Data Types

It is possible to discuss data types in terms of two sets of contrasting classifications. These are *scalar* versus *structured* types, and *primitive* versus *user–defined*. Scalar types are those in which values consist of indivisible entities, or scalars; typical examples are integers and floating point numbers. Structured types are those constructed from an aggregate of scalar or simpler structured types; examples of these are arrays and records. Primitive types are those that are supplied built–in to the programming language, and are almost always scalar types corresponding closely to machine primitives. User–defined types are those whose definitions must be supplied by the programmer; these are almost always structured types.

2.1.1 Scalar Data Types

Boolean data are those which have only two possible values, generally interpreted as *true* or *false*.

Integer types, some times called fixed point, implement a subset of the integer numbers, usually within some hardware defined range. Common ranges are -2^{15} to $2^{15}-1$ and -2^{31} to $2^{31}-1$. Occasionally a language will provide two different integer types, one with a smaller range that can result in some storage and computational efficiencies.

Real types, sometimes called floating point, are approximations to the real numbers of mathematics. They are approximations because machine hardware limits the precision with which they can be represented. As with integers, limits on the size of numbers are usually imposed, as are limits to the precision. Also, as with integers, it is common for a language to provide more than one precision, i.e., to allow a programmer to trade storage and computation costs for more accuracy of computation.

Character data types are those that can represent single characters of the character set of the underlying machine.

Enumerated data types are programmer declared, and consist of a small number of explicitly declared constants. Variables of enumerated types can therefore have only one of a small possible number of values.

2.1.2 Structured Data Types

Complex data are those that represent complex numbers. They are a structured type because they are represented by two scalar values, one each for the real and imaginary portions. In order for complex numbers to be convenient to use, they normally must be a primitive data type in the language.

Strings are sequences of zero or more characters.

Arrays are collections of objects, all of the same type, each indexed by a set of values. Crucial to arrays is the concept of order, where the location of one object relative to another is important.

Sets are collections of objects, all of the same type. In contrast to arrays, there is no implied ordering of objects, nor can two objects have the same value. Objects cannot be accessed by indexing.

Records are structured types whose components are accessed by a set of explicitly named selectors, rather than by indexing as is the case for arrays. Also in contrast to arrays, the components (or fields) of a record may be of different types.

2.2 Control Constructs

The previous section described, in general terms, some common types of data. This section deals with common types of control. At its most basic level, every language provides facilities so that the programmer may specify the *sequence*, *selection* and *repetition* of operations.

2.2.1 Sequence

Sequencing is the simplest type of control structure, and simply means that all of a set of statements are to be executed in the order written. As an example, consider the `BEGIN--END` statement, an explicit grouping construct that is often necessary to allow a number of statements to appear where the language syntax normally allows only one.

```
BEGIN
  S1;
  S2;
  ...
  Sn;
  END
```

Here, it is understood that the statements S_1 through S_n are to be executed consecutively, in the order shown.

2.2.2 Selection

Selection is the ability to test for certain conditions, and to conditionally execute one of a set of statements, depending on the outcome of this test.

In a language–independent manner, it is possible to distinguish three common selection mechanisms.

```
IF c
  THEN S1
  ELSE S2
```

The `IF-THEN-ELSE` construct is perhaps the simplest and most common form of selection. The condition c is tested; if it is true, statement S_1 is executed, otherwise statement S_2 is executed. It is normal for the `ELSE` part and associated statement S_2 to be optional.

```
CASE E0 OF
  E1 :  S1;
  E2 :  S2;
    ...
  En :  Sn;
  OTHERWISE S*
  END
```

In the CASE statement, expression E_0 is evaluated, and compared against the values of E_1 through E_n in turn. For the first j such that $E_0 = E_j$, the statement S_j, and only that statement, is executed. If no equality is found, the otherwise statement, S_*, is executed. It is common for the OTHERWISE clause and associated statement S_* to be optional.

NUMBEREDCASE E OF S_1; S_2; ...; S_n END

Expression E is evaluated and must produce an integer j such that $1 \leq j \leq n$; then statement S_j, and only that statement, is executed.

Note that only the IF-THEN-ELSE construct is a mandatory control construct. The other constructs can be simulated in terms of the IF statement. However, the two types of CASE construct are often provided because they assist the programmer and simplify and add clarity to a program.

2.2.3 Repetition

Two methods of obtaining repetition exist; these are *recursion* and *iteration*. Recursion is the ability of a procedure to specify invocation of itself (either directly or indirectly). No special syntax is used to indicate recursion.

Iteration is the repetitive execution of one or more statements until some condition is satisfied. There are several common ways of achieving iteration.

WHILE C DO S

The condition C of the WHILE is tested. If it is *true*, then the statement S is executed and the loop repeats until C is *false*. Presumably, the execution of S causes changes to data which will eventually result in C being *false*, otherwise the loop will never terminate. Note that S will not be executed at all, if C is *false* at the first test.

REPEAT S UNTIL C

Statement S is executed, then the condition C is tested. If it is *true*, the REPEAT statement will terminate. If it is *false*, S is executed again and the cycle continues. Note that S will always be executed at least once.

LOOP
  $S_1; \ldots; S_k;$
EXITIF C_1;
  $S_{k+1}; \ldots; S_l;$
EXITIF C_2;
  S_{l+1};
  ...;
  S_n;
END

Statements S_1 through S_n are executed indefinitely. The conditions on the EXITIF statements are tested in sequence with S_1 through S_n. The first C_i to evaluate to *true* causes the entire loop to terminate.

FOR $i := E_1$ TO E_2 DO S

In the FOR statement, expressions E_1 and E_2 are evaluated and must produce integral values. Variable i is initially set equal to E_1, and as long as $i \leq E_2$ statement S is executed. After each execution of S, the value of i is incremented by 1.

2.3 Program Organization

The previous two sections dealt with programs at the level of single statements or syntactic entities. This section presents the important concept of the aggregation of larger collections or assemblies of program statements. It is normal to distinguish two general levels of aggregation — *routines*, and *modules* and *programs*.

2.3.1 Routines

The first level of aggregation is the ability to name a collection of statements, to refer to these by name, and to cause their execution by referring to that name. The generic term for such a collection is a *routine*; specific names are usually *procedures*, *subroutines* and *functions*.

A procedure (or subroutine) is a named collection of statements, usually with parameters associated with the name so that different data may be used with the statements at different times. When a procedure is invoked, all of its useful work is accomplished through side–effect, i.e., through the modification of global variables or parameters or through input/output. Procedures do not return a value to the point of invocation. Syntactically, procedure invocation is by means of a separate statement.

A function is similar to a procedure, with the crucial difference that a function returns a value to the point of invocation. Syntactically, function invocation can be part of a larger expression or statement.

2.3.2 Modules and Programs

Often, two or more routines should logically be grouped together and thought of as a unit. This might be because they work on common data or implement similar or complimentary functions. When this happens, it is advantageous to group this routines together as a set. Such a grouping is called a module.

Finally, a program is a complete, self contained collection of routines and modules that completely specifies an implementation or process.

2.4 Other Attributes

Now that the more easily classified fundamentals have been discussed, this section addresses other important issues. These are issues that are hard to classify, either because there is rarely any specific syntax to support them, or because the concept itself is a little less well defined.

2.4.1 Compilation

Two such important issues are *separate* and *conditional* compilation. If separate compilation is supported, it is easier and more efficient to provide libraries of commonly needed routines. It is also easier to develop large programs because one can re–compile only those routines that are changed.

Conditional compilation refers to the ability of the compiler to make decisions about which statements may or may not be compiled in a routine. This allows the programmer to construct programs that are easily retargeted for different machines, or reconfigured in some manner. For example, debugging statements can be included in development versions, and excluded from production versions of programs.

2.4.2 Global Data and Modularity

When a module consists of several routines, it is often necessary that they share a common pool of data. At the same time, the routines of the module may not necessarily invoke each other, in which case the data cannot be passed through parameters. Those routines must then have access to data that is *global* to the routines in the module. On the other hand, to protect the data from inadvertent modification by other routines, the data should be local to the module itself.

2.4.3 Efficiency

Efficiency usually means the efficiency, in time or space, of *execution* of production programs. It is worthwhile to note that the concern for absolute efficiency is misplaced in many applications programs. In only rare cases, for the largest programs, and then only for parts of these, is execution efficiency of overriding importance.

Efficiency of execution is not necessarily inherent in a language; more important is the quality of the particular compiler. This is almost entirely dependent on the effort put into developing the compiler. Some language features , though, can influence the ease of development of compilers that produce efficient code. For example, the increment and decrement operators of C can make some types of looping more efficient. Array operators, as proposed for the new version of FORTRAN, could make some array manipulations more efficient, particularly on parallel or vector machines.

2.4.4 Input/Output

There are several issues regarding I/O in a language. One is whether I/O is accomplished via syntactically special statements, or is accomplished through library routines. This can influence the ease with which special applications are developed, with I/O that is accomplished through library routines generally being simpler and more flexible. Another issue is whether the I/O system is complete or not. That is, it should handle at least text and binary, sequential and random access. A third issue that is harder to quantify is the support that the I/O system provides for developing interactive programs. Particularly important to this last issue is the ease of dealing with input errors and other exceptional conditions.

2.4.5 Standardization

Official standards are the best way to assure some measure of portability of programs to other machines and operating systems. Therefore, standards are very important. Standards promote the implementation of compilers that all accept a common syntax, and that all interpret the statements in a common way. The standardization process itself can be beneficial, as it receives input from a large number of expert practitioners, and encourages much careful thought about language features.

3. Technical Comparisons

In this section, the four languages are compared with regard to the presence or absence of the attributes described in the preceding section, with explanatory notes where necessary. Although simply indicating the presence or absence of a capability is not enough, it does enable one to compare and assess the comprehensiveness and capabilities of languages. In the following discussion, it should be noted that a *yes* or *no* signifies whether the specified functionality exists by definition or concept in the language. The actual form or syntax may vary somewhat, however extensions to the language in the form of libraries, although they may achieve the desired functionality, are not considered here to be part of the fundamental definition.

Many references exist giving full details of each of the languages discussed. For further information, the reader is referred to: references [1] and [11] for FORTRAN; references [4] and [12] for C; references [2] and [10] for Pascal; and references [5], [14] and [18] for Modula–2. Other references exist that give a far more detailed technical comparison of programming languages. Of particular note are Coar [6] and Feuer et al. [8]. Unfortunately, FORTRAN is not well covered in these comparisons.

3.1 Data Types

	FORTRAN	C	Pascal	Modula–2
Boolean	yes	no(1)	yes	yes
Integer	yes	yes	yes	yes
Real	yes	yes	yes	yes
Extended Real	yes	yes	no	no
Character	yes	yes	yes	yes
Enumerated	no	yes	yes	yes
Complex	yes	no	no	no
Strings	yes	no(2)	no	no
Arrays	yes	yes	yes	yes
Sets	no	no	yes	yes
Records	no	yes	yes	yes

(1) C does not explicitly have a boolean type, but integers can be used in boolean expressions and can have the properties of booleans, with zero representing *false* and all other values representing *true*.

(2) In C, strings are represented by an array of characters, with the convention that a character value of zero indicates the end of the string. The language provides no string primitives, but relies on standard libraries to provide almost all of the functionality.

3.2 Control Constructs

	FORTRAN	C	Pascal	Modula–2
Sequence	yes	yes	yes	yes
IF–THEN	yes	yes	yes	yes
CASE	no	yes	yes	yes
NUMBEREDCASE	yes(1)	no	no	no
Recursion	no	yes	yes	yes
WHILE	no	yes	yes	yes
REPEAT	no	yes	yes	yes
LOOP	no	yes	no	yes
FOR	yes	yes(2)	yes	yes

(1) The computed `GOTO` of FORTRAN closely approximates the `NUMBEREDCASE` construct, but the syntax is quite different.

(2) C actually has a `FOR` loop that is much more general than the one described here, in which the programmer explicitly provides the starting assignment and terminating tests, as well as the increment procedure.

3.3 Program Organization

	FORTRAN	C	Pascal	Modula–2
Procedures	yes	yes(1)	yes	yes
Functions	yes	yes	yes	yes
Modules	no	no(2)	no	yes
Programs	yes	yes	yes	yes

(1) The only thing that distinguishes a procedure from a function in C, is the presence of a value expression after a return statement in the routine. It is expected that all routines will return a value, but that value may be ignored.

(2) In C, a programmer may optionally localize data and procedure declarations to a single source file, hiding them from other source files. This provides most of the functionality of modules.

3.4 Other Attributes

	FORTRAN	C	Pascal	Modula–2
Sep. Compilation	yes	yes	no	yes
Cond. Compilation	no	yes	no	no
Global Data	yes(1)	yes	yes	yes
Built–in I/O	yes	no	yes	no
Binary I/O	yes	no(2)	yes	no(2)
Random I/O	yes	no(2)	no	no(2)
Interactive I/O	no	no(2)	no	no(2)
Standardized	yes	no(3)	yes	no

(1) The common blocks of FORTRAN do allow routines to access global data, however that data cannot be isolated to a particular set of procedures, nor protected from use by other procedures.

(2) As I/O is implemented by library functions, and is not part of the language, whether these features are provided or not depends on the implementation of the available library.

(3) Work on official standards for C is in progress.

3.5 Summary

All of the attributes mentioned above are in some manner beneficial to the program development process, but not all to the same extent. Therefore, a language that has more of the attributes is in some sense a "better" language, although this certainly must be evaluated in the context of a particular application task.

There are many other factors involved in comparing languages. Some of these are convenience of use, programmer experience, existing software base, and safety. To summarize, the attributes mentioned above are important and must be considered, but there are many other important factors that cannot be adequately discussed here. The intent of the above discussion has been to touch on the most crucial and fundamental aspects, and to emphasize that a practicing engineer must be knowledgeable of these.

4. Observations and Recommendations

While it is not advisable to make specific recommendations, this final section discusses some areas in which each language is strongest, and why that is the case.

FORTRAN certainly has been the dominant language in the traditional engineering applications areas, such as finite element and other numerically intensive applications. This is partially due to historical inertia and to the fact that most engineers know only FORTRAN. However, FORTRAN has always done a reasonable job at supporting purely numerical computation and there have been large libraries of routines available for many years. FORTRAN may support double precision numeric computation slightly better than the other languages, and it is the only one of the languages mentioned that supports complex arithmetic. Therefore, to develop an application that is predominately numerical in nature, FORTRAN is still a top choice, particularly where use of math libraries, double precision, or complex arithmetic is involved. However, one must be careful; it is often surprising how much non–numerical programming and data management is involved in what appear to be numerical applications.

C has great strength in areas where the management of data, is a predominate action, as is the case in many applications. Applications programs in which this is so include graphics programs, those that manipulate databases, those in which man–machine interaction is important, and CAD programs, such as drafting, or design aids. In addition, the style of applications program is evolving toward the use of tools — individually small, relatively simple programs that are integrated in some manner into a system [9]. This again is an area where C is at an advantage due to its emphasis on data manipulation. Informal surveys have shown that the popularity of C is growing rapidly in engineering programming, and it is expected that this trend will accelerate.

Pascal is generally not recommended for the development of large programs or systems. Being a teaching language, it has very serious deficiencies for developing large programs. However it is very strong with respect to data manipulation and is an adequate language for the smaller programs that are becoming more common. The official standardization process is attempting to deal with the shortcomings of the language and that may positively influence its future as an engineer's language. In addition, because of its popularity as an introductory language in many universities, many graduating engineers now know that language. Therefore, it is likely to remain an important language for years to come.

Because Modula–2 is so new, with few high quality compilers yet available, and no official standard yet on the horizon, one should be very cautious about choosing this language for extensive work at this time. However, several factors combine to ensure that Modula–2 will be very important in the future. The first is its similarity to Pascal and the fact that Pascal is such a popular teaching language. Because of this, many graduating engineers will know it or will be able to learn it very easily. In addition, it is a well–designed language with good, abundant features, yet without the complexity of some other languages. These factors indicate that Modula–2 will become the language of choice for many applications.

Acknowledgements

Portions of this work were supported by the Natural Science and Engineering Research Council of Canada under grant A1419. Portions were also supported by the National Science Foundation under grants MSM–8451465, a Presidential Young Investigator award, and CEE–8319869, "Generative Database Systems for Structural Engineering Design."

References

[1] American National Standards Committee, *American National Standard Programming Language FORTRAN* (FORTRAN 77), Document X3J3/90, X3 Secretariat, CBEMA/-Standards, 1828 L Street, N.W., Washington, DC, 20036, (1978).

[2] L. Atkinson, *Pascal Programming*, Wiley, Chichester, UK, (1981).

[3] J. Backus, "The History of FORTRAN I, II and III," in R. L .Wexelblat (ed.), *History of Programming Languages*, Academic Press, New York, 1981.

[4] BYTE Magazine, *Special Issue on C*, August (1983).

[5] BYTE Magazine, *Special Issue on Modula–2*, August (1984).

[6] D. Coar, "Pascal, Ada, and Modula–2: A system programmer's comparison," *BYTE*, August, (1984).

[7] M. Elson, *Concepts of Programming Languages*, Science Research Associates, Chicago (1973).

[8] A. Feuer and N. Gehani (eds.), *Comparing and Assessing Programming Languages: Ada, C and Pascal*, Prentice–Hall, Englewood Cliffs, NJ (1984).

[9] N. Holtz and W. Rasdorf, "LISP – A CAD System Programming Language," *Journal of the Technical Topics in Civil Engineering*, ASCE, 109(1), 1983, pp. 58–72.

[10] K. Jensen and N. Wirth, *PASCAL User Manual and Report, 2nd ed.*, Springer–Verlag, New York (1978).

[11] H. Katzan, *FORTRAN 77*, Van Nostrand Reinhold. New York (1978).

[12] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice–Hall, Englewood Cliffs, NJ (1978).

[13] B. J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart and Winston, New York (1983).

[14] G. Pomberger, *Software Engineering and Modula–2*, Prentice–Hall International, Englewood Cliffs, NJ (1984).

[15] W. J. Rasdorf and O. O. Storaasli, "Educational Fundamentals of Computer Aided Engineering," *International Journal of Applied Engineering Education*, Pergamon Press, in press.

[16] M. Shaw, G. T. Almes, J. N. Newcomer, B. K. Reid and W. A. Wulf, "A Comparison of Programming Languages for Software Engineering," in A. Feuer and N. Gehani (eds.), *Comparing and Assessing Programming Languages: Ada, C and Pascal*, Prentice–Hall, Englewood Cliffs, NJ (1984).

[17] R. L .Wexelblat (ed.), *History of Programming Languages*, Academic Press, New York, 1981.

[18] N. Wirth, *Programming in Modula–2, 2nd ed.*, Springer–Verlag, Berlin (1982).

[19] W. A. Wulf, M. Shaw, P. N. Hilfinger, and L. Flon, *Fundamental Structures of Computer Science*, Addison–Wesley, Reading, MA., 1981.