A Programming Language

Kenneth E. Iverson

To My Many Teachers

PREFACE

Applied mathematics is largely concerned with the design and analysis of explicit procedures for calculating the exact or approximate values of various functions. Such explicit procedures are called algorithms or *programs.* Because an effective notation for the description of programs exhibits considerable syntactic structure, it is called a *programming language.*

Much of applied mathematics, particularly the more recent computer-related areas which cut across the older disciplines, suffers from the lack of an adequate programming language. It is the central thesis of this book that the descriptive and analytic power of an adequate programming language amply repays the considerable effort required for its mastery. This thesis is developed by first presenting the entire language and then applying it in later chapters to several major topics.

The areas of application are chosen primarily for their intrinsic interest and lack of previous treatment, but they are also designed to illustrate the universality and other facets of the language. For example, the microprogramming of Chapter 2 illustrates the divisibility of the language, i.e., the ability to treat a restricted area using only a small portion of the complete language. Chapter 6 (Sorting) shows its capacity to compass a relatively complex and detailed topic in a short space. Chapter 7 (The Logical Calculus) emphasizes the formal manipulability of the language and its utility in theoretical work.

The material was developed largely in a graduate course given for several years at Harvard and in a later course presented repeatedly at the IBM Systems Research Institute in New York. It should prove suitable for a two-semester course at the senior or graduate level. Although for certain audiences an initial presentation of the entire language may be appropriate, I have found it helpful to motivate the development by presenting the minimum notation required for a given topic, proceeding to its treatment (e.g., microprogramming), and then returning to further notation. The 130-odd problems not only provide the necessary finger exercises but also develop results of general interest.

Chapter 1 or some part of it is prerequisite to each of the remaining ”applications” chapters, but the applications chapters are virtually independent of one another. A complete appreciation of search techniques (Chapter 4) does, however, require a knowledge of methods of representation (Chapter 3). The cross references which do occur in the applications chapters are either nonessential or are specific to a given figure, table, or program. The entire language presented in Chapter 1 is summarized for reference at the end of the book.

In any work spanning several years it is impossible to acknowledge adequately the many contributions made by others. Two major acknowledgments are in order: the first to Professor Howard Aiken, Director Emeritus of the Harvard Computation Laboratory, and the second to Dr. F.P. Brooks, Jr. now of IBM.

It was Professor Aiken who first guided me into this work and who provided support and encouragement in the early years when it mattered. The unusually large contribution by Dr. Brooks arose as follows. Several chapters of the present work were originally prepared for inclusion in a joint work which eventually passed the bounds of a single book and evolved into our joint Automatic Data Processing and the present volume. Before the split, several drafts of these chapters had received careful review at the hands of Dr. Brooks, reviews which contributed many valuable ideas on organization, presentation, and direction of investigation, as well as numerous specific suggestions.

The contributions of the 200-odd students who suffered through the development of the material must perforce be acknowledged collectively, as must the contributions of many of my colleagues at the Harvard Computation Laboratory. To Professor G.A. Salton and Dr. W.L. Eastman, I am indebted for careful reading of drafts of various sections and for comments arising from their use of some of the material in courses. Dr. Eastman, in particular, exorcised many subtle errors from the sorting programs of Chapter 6. To Professor A.G. Oettinger and his students I am indebted for many helpful discussions arising out of his early use of the notation. My debt to Professor R.L. Ashenhurst, now of the University of Chicago, is apparent from the references to his early (and unfortunately unpublished) work in sorting.

Of my colleagues at the IBM Research Center, Messrs. L.R. Johnson and A.D. Falkoff, and Dr. H. Hellerman have, through their own use of the notation, contributed many helpful suggestions. I am particularly indebted to L.R. Johnson for many fruitful discussions on the applications of trees, and for his unfailing support.

On the technical side, I have enjoyed the assistance of unusually competent typists and draughtsmen, chief among them being Mrs. Arthur Aulenback, Mrs. Philip J. Seaward, Jr., Mrs. Paul Bushek, Miss J.L. Hegeman, and Messrs. William Minty and Robert Burns. Miss Jacquelin Sanborn provided much early and continuing guidance in matters of style, format, and typography. I am indebted to my wife for assistance in preparing the final draft.

Kenneth E. Iverson

May, 1962
Mount Kisco, New York

Contents

Chapter 1

The Language

1.1 Introduction

Applied mathematics is concerned with the design and analysis of algorithms or programs. The systematic treatment of complex algorithms requires a suitable programming language for their description, and such a programming language should be concise, precise, consistent over a wide area of application, mnemonic, and economical of symbols; it should exhibit clearly the constraints on the sequence in which operations are performed; and it should permit the description of a process to be independent of the particular representation chosen for the data.

Existing languages prove unsuitable for a variety of reasons. Computer coding specifies sequence constraints adequately and is also comprehensive, since the logical functions provided by the branch instructions can, in principle, be employed to synthesize any finite algorithm. However, the set of basic operations provided is not, in general, directly suited to the execution of commonly needed processes, and the numeric symbols used for variables have little mnemonic value. Moreover, the description provided by computer coding depends directly on the particular representation chosen for the data, and it therefore cannot serve as a description of the algorithm per se.

Ordinary English lacks both precision and conciseness. The widely used Goldstine-von Neumann (1947) flowcharting provides the conciseness necessary to an over-all view of the processes, only at the cost of suppressing essential detail. The so-called pseudo-English used as a basis for certain automatic programming systems suffers from the same defect. Moreover, the potential mnemonic advantage in substituting familiar English words and phrases for less familiar but more compact mathematical symbols fails to materialize because of the obvious but unwonted precision required in their use.

Most of the concepts and operations needed in a programming language have already been defined and developed in one or another branch of mathematics. Therefore, much use can and will be made of existing notations. However, since most notations are specialized to a narrow field of discourse, a consistent unification must be provided. For example, separate and conflicting notations have been developed for the treatment of sets, logical variables, vectors, matrices, and trees, all of which may, in the broad universe of discourse of data processing, occur in a single algorithm.

1.2 Programs

A *program statement* is the specification of some quantity or quantities in terms of some finite operation upon specified operands. Specification is symbolized by an arrow directed toward the specified quantity. thus ”y is specified by sin x” is a statement denoted by

```
y ← sin x.
```

A set of statements together with a specified order of execution constitutes a *program.* The program is finite if the number of executions is *finite.* The *results* of the program are some subset of the quantities specified by the program. The *sequence* or order of execution will be defined by the order of listing and otherwise by arrows connecting any statement to its successor. A cyclic sequence of statements is called a *loop.*

(TODO: FIGURES 1.1 AND 1.2)

Thus Program 1.1 is a program of two statements defining the result v as the (approximate) area of a circle of radius x, whereas Program 1.2 is an infinite program in which the quantity z is specified as $(2y)^n$ on the n-th execution of the two statement loop. Statements will be numbered on the left for reference.

A number of similar programs may be subsumed under a single more general program as follows. At certain *branch points* in the program a finite number of alternative statements are specified as possible successors. One of these successors is chosen according to criteria determined in the statement or statements preceding the branch point. These criteria are usually stated as a *comparison* or test of a specified relation between a specified pair of quantities. A branch is denoted by a set of arrows leading to each of the alternative successors, with each arrow labeled by the comparison condition under which the corresponding successor is chosen. The quantities compared are separated by a colon in the statement at the branch point, and a labeled branch is followed if and only if the relation indicated by the label holds when substituted for the colon. The conditions on the branches of a properly defined program must be disjoint and exhaustive.

Program 1.3 illustrates the use of a branch point. Statement α5 is a comparison which determines the branch to statements β1, δ1, or γ1, according as $z > n$, $z = n$, or $z < n$. The program represents a crude by effective process for determining $x = n^{\frac{2}{3}}$ for any positive cube n.

(TODO: FIGURE 1.3)

Program 1.4 shows the preceding program reorganized into a compact linear array and introduces two further conventions on the labeling of branch points. The listed successor of a branch statement is selected if none of the labeled conditions is met. Thus statement 6 follows statement 5 if neither of the arrows (to exit or to statement 8) are followed, i.e. if $z < n$. Moreover, any unlabeled arrow is always followed; e.g., statement 7 is invariably followed by statement 3, never by statement 8.

A program begins at a point indicated by an *entry arrow* (step 1) and ends at a point indicated by an *exit arrow* (step 5). There are two useful consequences of confining a program to the form of a linear array: the statements may be referred to by a unique serial index (statement number), and unnecessarily complex organization of the program manifests itself in crossing branch lines. The importance of the latter characteristic in developing clear and comprehensible programs is not sufficiently appreciated.

(TODO: FIGURES 1.4 AND 1.5)

A process which is repeated a number of times is said to be iterated, and a process (such as in Program 1.4) which includes one or more iterated subprocesses is said to be iterative. Program 1.5 shows an iterative process for the matrix multiplication

```
C ← AB
```

defined in the usual way as

$$C^i_j = \sum_{k=1}^{v(\mathbf{A})} A^k_i \times B^k_j \qquad \begin{array}{c} i = 1, 2, ..., \mu(\mathbf{A}), \\ j = 1, 2, ..., v(\mathbf{B}), \end{array}$$

where the dimensions of an $m \times n$ matrix X (of m rows and n columns) is denoted by $\mu(X) \times v(X)$.

Program 1.5. Steps 1-3 initialize the indices, and the loop 5-7 continues to add successive products to the partial sum until k reaches zero. When this occurs, the process continues through step 8 to decrement j and to repeat the entire summation for the new value of j, providing that it is not zero. If j is zero, the branch to step 10 decrements i and the entire process over j and k is repeated from $j = v(\mathbf{B})$, providing that i is not zero. If i is zero, the process is complete, as indicated by the exit arrow.

In all examples used in this chapter, emphasis will be placed on clarity of description of the process, and considerations of efficient execution by a computer or class of computers will be subordinated. These considerations can often be introduced later by relatively routine modifications of the program. For example, since the execution of a computer operation involving an indexed variable is normally more costly than the corresponding operation upon a nonindexed variable, the substitution of a variable s for the variable C^i_j specified by statement 5 of Program 1.5 would accelerate the execution of the loop. The variable s would be initialized to zero before each entry to the loop and would be used to specify C^i_j at each termination.

The practice of first setting an index to its maximum value and then decrementing it (e.g., the index k in Program 1.5) permits the termination comparison to be made with zero. Since zero often occurs in comparisons, it is convenient to omit it. Thus, if a variable stands alone at a branch point, comparison with zero is implied. Moreover, since a comparison on an index frequently occurs immediately after it is modified, a branch at the point of modification will denote branching upon comparison of the indicated index with zero, the comparison occurring *after* modification. Designing programs to execute decisions immediately after modification of the controlling variable results in efficient execution as well as notational elegance, since the variable must be present in a central register for both operations.

Since the sequence of execution of statements is indicated by connecting arrows as well as by the order of listing, the latter can be chosen arbitrarily. This is illustrated by the functionally identical Programs 1.3 and 1.4. Certain principles of ordering may yield advantages such as clarity or simplicity of the pattern of connections. Even though the advantages of a particular organizing principle are not particularly marked, the uniformity resulting from its consistent application will itself be a boon. The scheme here adopted is called the *method of leading decisions*: the decision on each parameter is placed as early in the program as practicable, normally just before the operations indexed by the parameter. This arrangement groups at the head of each iterative segment the initialization, modification, and the termination test of the controlling parameter. Moreover, it tends to avoid program flaws occasioned by unusual values of the argument.

(TODO: FIGURE 1.6)