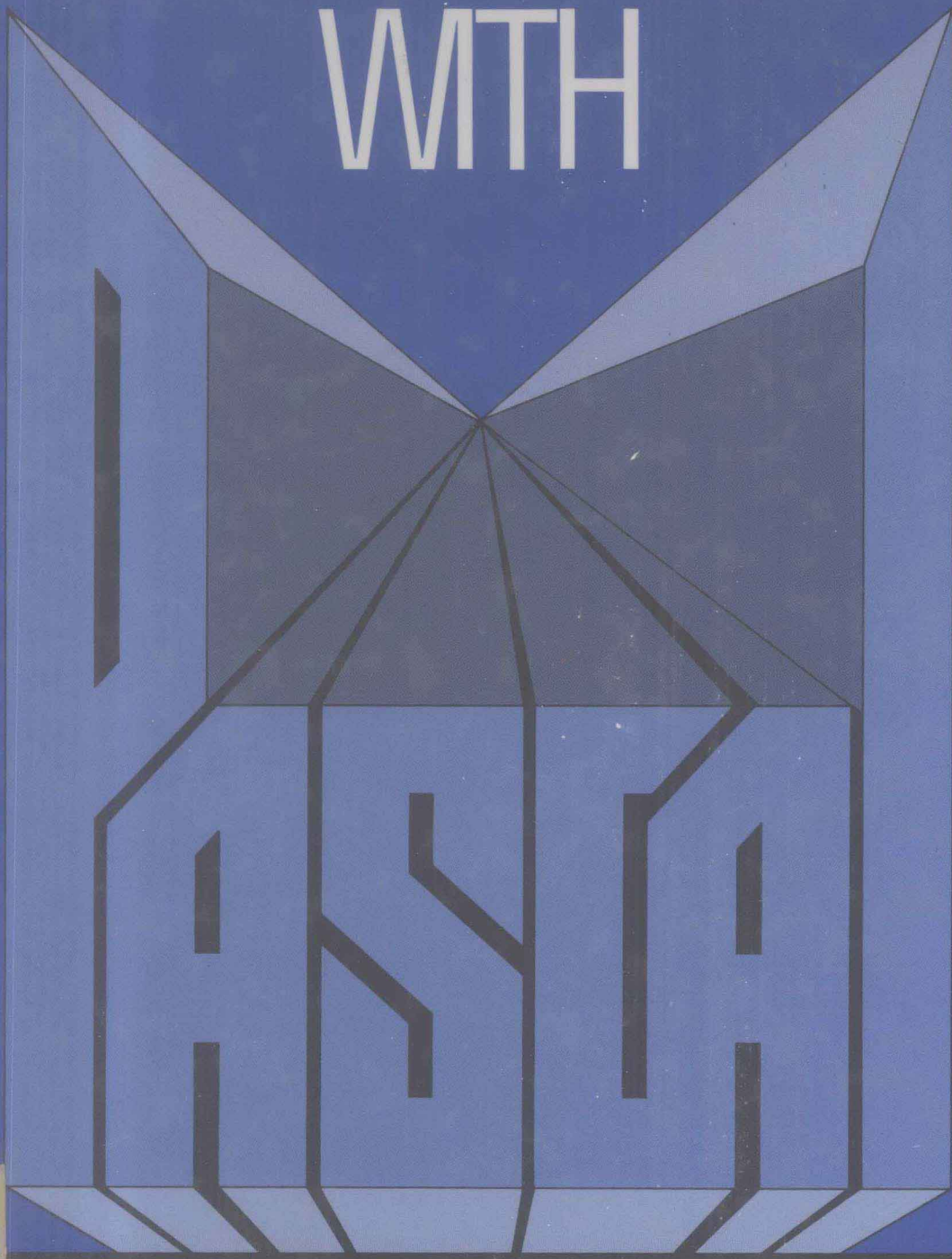
COMPUTABILITY WITH BASIC



John S. Mallozzi / Nicholas J. De Lillo

COMPUTABILITY WITH PASCAL

JOHN S. MALLOZZI
Iona College

NICHOLAS J. DE LILLO
Manhattan College

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Mallozzi, John S. (date)
Computability with Pascal.

Bibliography: p.
Includes index.
1. Computable functions—Data processing. 2. Recursive functions—Data processing. 3. PASCAL (Computer program language) I. De Lillo, Nicholas J. II. Title.
QA9.59.M34 1984 511 83-24450
ISBN 0-13-164443-2

Editorial/production supervision: *Raeia Maes*
Cover design: *Ben Santora*
Manufacturing buyer: *Gordon Osbourne*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-164443-2

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

COMPUTABILITY WITH PASCAL

To
MaryAnn

To My Parents

Anna and Rocco De Lillo

PREFACE

The first course in theoretical computer science presents a challenging pedagogical problem. Students who have become result and performance oriented must accept the need to study concepts that appear to have little, if any, applicability to practical computing, and whose implementation is, if not impossible, extremely inefficient. In this text we attack the problem in two ways.

First, the level of presentation is sufficiently elementary so that students may study the subject earlier in their training. This permits integration of the desired point of view into the basic component of the student's education in computer science.

Second, the Pascal programming language is used throughout the text. This is done in two very different ways. In some settings, such as the study of finite automata, simulators written in Pascal bring the desired concreteness, without trivializing examples. The second use of Pascal is more formal. Informal consideration of the distinction between the static idea of a program and the dynamic idea of a computation shows the student the need both to avoid the full power of a language like Pascal and to escape the restrictions of actual implementations. Thus motivated, we introduce "limited Pascal," which is a carefully chosen subset of Pascal, in an implementation-free setting. Precise definitions are stated in terms of this subset.

Thus, at a more elementary level than in other texts, the theory of computation is presented in a concrete setting, with the Pascal language as both a tool and, in a restricted and abstracted form, a part of the theory. This book may be used as a text to satisfy CS16 of the ACM's Curriculum '78.

The text draws in the reader as a participant through running Pascal programs and establishing some of the theoretical results. Both these goals are established

by the exercises, which also provide needed reinforcement of concepts through concrete applications, along the lines of the examples in the body of the text.

The authors wish to express their gratitude to Professor Thomas Smith, Chairperson of the Department of Mathematics and Computer Science, Manhattan College, and Professor Catherine Ricardo, Chairperson of the Department of Computer Science, Iona College, for permitting class-testing of much of the material; to Mr. Antony Halaris and Ms. Lynda Sloan, and their staff in the Iona College Computing Center, for providing the high level of support and superb facilities which contributed to the manuscript in ways ranging from running programs to text editing. We also thank our students, who were sometimes exposed to very preliminary versions of this material.

The second author wishes to express his gratitude to his professor and mentor, Martin Davis, for having first introduced him to the beauty of the theory of recursive functions and effective computability.

We also want to thank the following individuals, who provided encouragement during the rewriting of preliminary versions of the work: F. Beckman and K. McAloon, Brooklyn College, CUNY; C. Colbourn, University of Saskatchewan; M. Fitting, Lehman College, CUNY; and D. Prener, IBM. We express our gratitude also to the editorial staff at Prentice-Hall, notably J. Fegen, J. Wait, L. Frankel and R. Maes, who supervised the work from its earliest stages to the final form presented here.

Finally, we wish to thank our wives and children, and other members of our respective families, for their understanding and emotional fortitude and endurance throughout the development of this book.

John S. Mallozzi
Nicholas J. De Lillo

COMPUTABILITY WITH PASCAL

CONTENTS

1

PASCAL-COMPUTABLE FUNCTIONS

This chapter introduces some of the basic concepts that will be used in this book, including, in particular, *computation* and *Pascal-computable* functions and *primitive recursive* functions.

1.1 PRELIMINARY CONCEPTS

The functions we will study operate on and have values that are non-negative integers. Thus, we establish the following conventions:

- *Number* means non-negative integer.
- *Function* means function of number variables with number values.

A Pascal-computable function may be defined informally as a function $y = f(x_1, x_2, \ldots, x_n)$, such that there exists a Pascal program PROG which when executed with input any choice of values for the x_i produces output $y = f(x_1, x_2, \ldots, x_n)$.

Example 1.1.1

```
PROGRAM FIRST (INPUT, OUTPUT);
VAR
  X, Y : INTEGER;
BEGIN
  READ (X);
  Y := 2 * X + 3;
  WRITELN (Y)
END.
```

If this program is executed with any number x contained in file INPUT, then, upon termination, the file OUTPUT will contain the number $2x + 3$.

The fact that the program also works for negative values is of no concern to us. What is important is that it is not accurate to say that "all" values of $f(x) = 2x + 3$ are computed by this program. Given any implementation of Pascal, there is a MAXINT (the largest allowable value for variables of type INTEGER). If, for instance, we set x = MAXINT, then $f(x)$ is perfectly well-defined, but the program will not work. For now, we do not deal with this problem, being content to point out that, *in principle*, the program is correct for any input; any limitation is "physical." (Note that the program makes no reference to MAXINT and that a literal must be input.) We shall return to this question later.

We now look at a program that computes a function of several variables.

Example 1.1.2

```
PROGRAM SUM (INPUT, OUTPUT);
VAR
  X1, X2, Y : INTEGER;
BEGIN
  READ (X1, X2);
  Y := X1 + X2;
  WRITELN (Y)
END.
```

When there is more input, it is convenient to use arrays, as shown below.

Example 1.1.3

```
PROGRAM SIGMA (INPUT, OUTPUT);
CONST
  N = n; (* Choose an appropriate value for n *)
VAR
  X : ARRAY [1 .. N] OF INTEGER;
  I, Y : INTEGER;
BEGIN
  Y := 0;
  FOR I := 1 TO N DO
    BEGIN
      READ (X[I]);
      Y := Y + X[I]
    END;
  WRITELN (Y)
END.
```

We assume that N and the number of input values match and that the input is presented properly, so that no READ error occurs. In the theory, we use no arrays. Thus, when a concept is defined for a sequence $x_1, x_2, \ldots, x_n$, we understand that, for convenience, we may choose to implement the concept using arrays.

A *program* is a *static* object: It is a list of declarations and statements that specifies objects and actions *to be* taken when the program is executed. A *computation*, as we shall define it here, is *dynamic:* It is what is actually done when a program is executed on a particular input list. If we "freeze" a computation in progress, we obtain a *snapshot*, or *instantaneous description*. We shall make these concepts precise in Section 1.2; for now, we introduce them informally by way of examples.

Example 1.1.4

Consider PROGRAM FIRST of Example 1.1.1, with input 3. If we execute this program on the given input, the computation obtained may be pictured as follows:

```
3 =>
      (1)  X <-  , Y <-      READ (X)
      (2)  X <- 3, Y <-      Y := 2*X + 3
      (3)  X <- 3, Y <- 9    WRITELN (Y)
      (4)  X <- 3, Y <- 9    .
                                          => 9
```

On the first line we have an *input list* (here containing only one item); on the last line we have an *output value*. The numbered statements are *snapshots*. Each snapshot contains a list of all variables declared in the program, with the then-current values of those variables (if any) on the right of the arrow. We call this list a *state* of the computation. Next to the state is the *next instruction* to be executed. The *initial snapshot* has the first instruction, and the *terminal snapshot* has only a period to indicate that we have reached the end of the program.

Note that BEGIN, END and ; do not appear because they are punctuation symbols, not instructions. Note also that until a value has been either assigned to or READ into a variable, we cannot assume any particular value for that variable. Some systems automatically preassign a value of 0; others give "garbage"; still others consider an attempt to access such a value an error. Although the last approach is most in the "spirit of Pascal," we do not assume it to be used in our implementation. Thus, we leave the value slot blank initially and (a good programming practice in any case!) never reference an uninitialized variable.

Example 1.1.5

```
PROGRAM DIVISOR (INPUT, OUTPUT);
VAR
  X, Y : INTEGER;
BEGIN
  READ (X);
  IF X <= 1 THEN
    Y := 1
  ELSE
```

```
    BEGIN
      Y := 2;
      WHILE X MOD Y <> 0 DO
        Y := Y + 1
    END;
  WRITELN (Y)
END.
```

If we execute this program with input list 9, the following computation results:

```
9 =>
    (1)  X <- , Y <-     READ (X)
    (2)  X <- 9, Y <-    X <= 1 ?
    (3)  X <- 9, Y <-    Y := 2
    (4)  X <- 9, Y <- 2  X MOD Y <> 0 ?
    (5)  X <- 9, Y <- 2  Y := Y + 1
    (6)  X <- 9, Y <- 3  X MOD Y <> 0 ?
    (7)  X <- 9, Y <- 3  WRITELN (Y)
    (8)  X <- 9, Y <- 3  .
                                        => 3
```

Note that snapshots (4) through (6) correspond to a single statement (WHILE). To execute a WHILE statement, a test must be performed to decide whether or not to execute the DO statement. We indicate the tests (by "?") explicitly, even though they are not explicit in the program. (They are explicit, of course, on the machine-language level.) We see, therefore, that the relationship between "instruction" and "statement" is not one-to-one. The concepts we have referred to above must be made precise: This is the task of Section 1.2.

EXERCISES 1.1

Unless we specify otherwise, computable will mean Pascal-computable.

1. Write a program that computes the function $f(x) = 3x + 5$.
2. Write a program that computes the product function $f(x_1, x_2) = x_1 \cdot x_2$.
3. Write a program that computes the product of n numbers. (Use arrays, with $N = 5$). Discuss the practical limitations on such a function.
4. Trace the computation that results from PROGRAM SUM with input list 7 5. (See Example 1.1.2.)
5. Using input 4, trace the computation that results from the program written for Exercise 1.
6. Using the input list 2 3, trace the computation that results from the program written for Exercise 2.
7. Using the input list 2 4 6 8 10, trace the computation that results from PROGRAM SIGMA of Example 1.1.3.

1.2 PASCAL COMPUTABILITY

We now turn to more precise descriptions of the concepts introduced in Section 1.1. A *state* of a program, PROG, is a list of items of the following form:

```
variable <- value
```

where variable is a declared variable of PROG and value is either a possible value of that variable or is empty.

Example 1.2.1

Each of the following has the form of a state of PROGRAM DIVISOR (Example 1.1.5):

```
X <- 9, Y <- 3
and X <- 9, Y <- 10.
```

Since only the first can actually occur, only it is a state.

An *instruction* is an action taken to execute a statement. Essentially, we want an instruction to be something we can conveniently think of as a single action. This may or may not correspond exactly to execution of one statement. Let us look at a few more examples.

Example 1.2.2

```
        STATEMENT              INSTRUCTION(S)
a)      Y := 2*X + 3           Y := 2*X + 3
b)      IF X < 3 THEN          1) X < 3 ?
          Y := 2*X             2) Y := 2*X, or none
                                  depending on answer.
```

Thus, a statement may correspond to one or more instructions. As (b) above shows, particular executions of PROG may have different instructions. The problem here is the vagueness of the terms "instruction" and "action." Pascal is sufficiently expressive that a complete description would be tedious. Instead, we look at a *subset* of Pascal, which we shall refer to as *limited Pascal*. The subset is small enough so that we can give a complete description of "instruction," but, at the same time, large enough so that we can write programs with minimal difficulty. Later, we shall discuss just how "limited" this subset is.

We define *limited Pascal* as containing the following, and only these, concepts from the Pascal language:

- PROGRAM
- Types INTEGER and BOOLEAN and VAR declarations for these types

- Assignment for INTEGER and BOOLEAN variables
- READ, READLN, WRITE, WRITELN for integers; WRITE,WRITELN for BOOLEAN
- Arithmetic for INTEGER (+,−,*,DIV,MOD); logical operations for BOOLEAN (OR,AND,NOT)
- Comparisons of integers (<,>,=,etc.); testing of BOOLEAN variables (TRUE or FALSE)
- BEGIN-END
- IF-THEN-ELSE
- WHILE

It should be clear that we could limit this further without great difficulty. For example, Boolean variables can be replaced by integers (1 for TRUE, 0 for FALSE). On the other hand, it is fairly simple to define other Pascal concepts in terms of the ones we use. In particular, if *A* and *B* are fixed, then

```
FOR I := A TO B DO
  BEGIN
    . . . (A and B unchanged) . . .
  END
```

can be written

```
I := A;
WHILE I <= B DO
  BEGIN
    . . . . . .
  I := I + 1
  END
```

Now let us describe instructions for each of the concepts of limited Pascal:

1. Instructions generated by a compound statement are generated by the individual component statements in sequence.
2. The instructions corresponding to assignment, input, or output statements are the statements themselves. Any expressions are written as is.
3. The instructions corresponding to IF condition THEN statement1 ELSE statement2 are as follows:

   condition ? statement1 or condition ? statement2

   depending on the outcome of the test of condition. (Recall that we are discussing *computations* here—a program may yield different instructions for different executions). Here, "condition" is either Boolean or an integer comparison. The statements may be compound, using BEGIN-END, in which case the entire sequence is included.