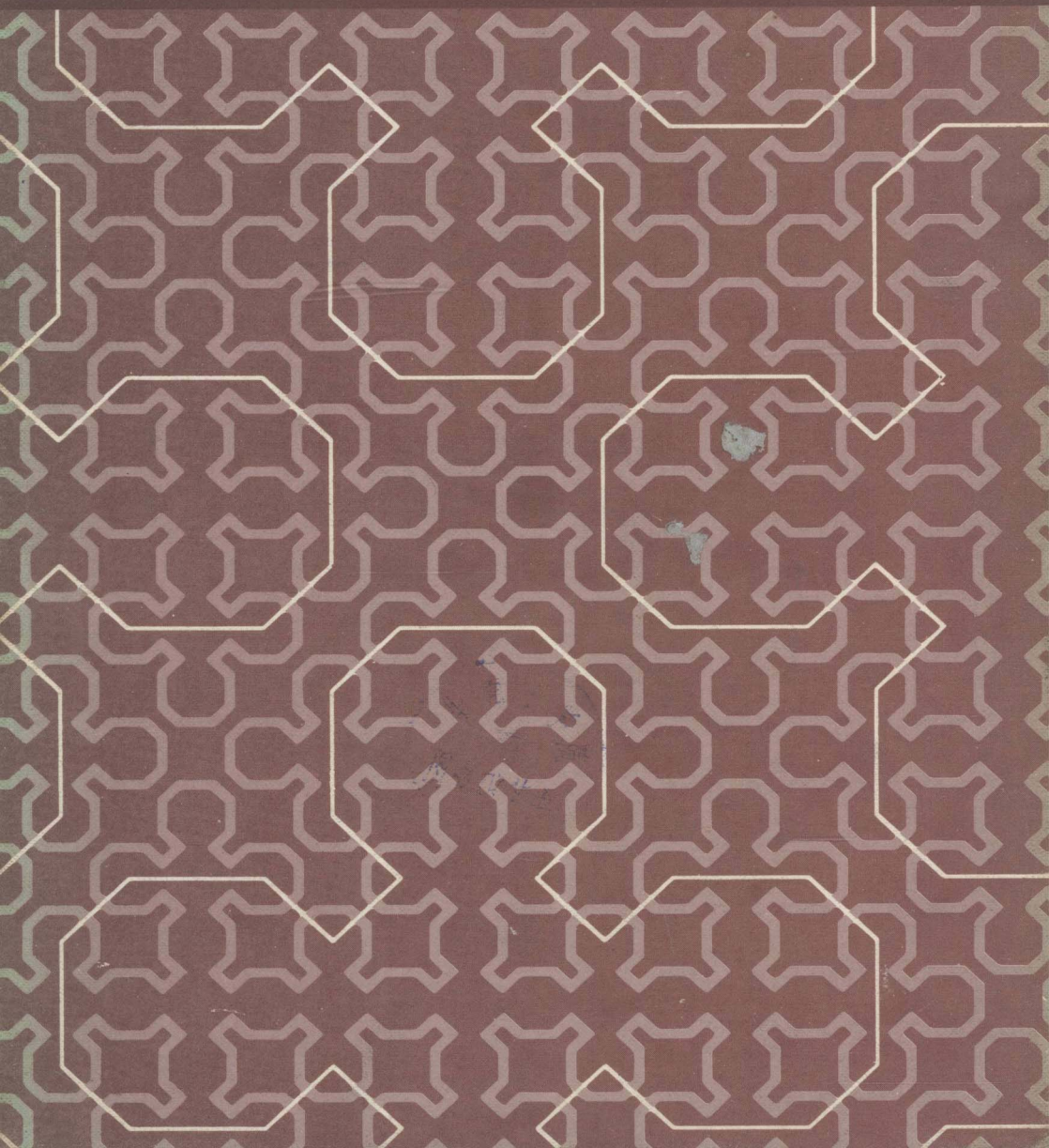
Cambridge Computer Science Texts · 8

Algol 68
a first and second course

ANDREW D. McGETTRICK



ALGOL 68

a first and second course

ANDREW D. McGETTRICK

Department of Computer Science
University of Strathclyde
Glasgow, Scotland

CAMBRIDGE UNIVERSITY PRESS
CAMBRIDGE
LONDON • NEW YORK • MELBOURNE

Published by the Syndics of the Cambridge University Press
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
Bentley House, 200 Euston Road, London NW1 2DB
32 East 57th Street, New York, NY 10022, USA
296 Beaconsfield Parade, Middle Park, Melbourne 3206, Australia

First published 1978

Printed in Great Britain at the University Press, Cambridge

Library of Congress cataloguing in publication data
McGettrick, Andrew D. 1944-
ALGOL 68: a first and second course
(Cambridge computer science texts; 8)
Includes index
1. ALGOL (Computer program language) I. Title
II. Series
QA76.73.A24M3 001.6'424 77-1104
ISBN 0 521 21412 2 hard covers
ISBN 0 521 29143 7 paperback

ALGOL 68

To my wife, Sheila

PREFACE

This book originated from lectures first given at the University of Strathclyde in 1973–4 to first year undergraduates, many of whom had no previous knowledge of programming. Many of the students were not taking computer science as their main subject but merely as a subsidiary subject. They therefore served as a suitable audience on whom to inflict lectures attempting to teach ALGOL 68 as a first programming language.

The book itself is concerned with the revised version of ALGOL 68 (see Acta Informatica, vol. 5, Fasc 1–3, 1975, pp. 1–236). It consists of nine chapters. I believe that, for a first course on programming, the material contained in chapters 1–5 is suitable; this forms an ALGOL-60-type-subset. The more advanced features of ALGOL 68 are contained in the later chapters and these provide suitable material for a second course. The individual chapters (chapters 6–9) are for the most part self-contained. Appendix A summarises in a convenient way the ALGOL 68 standard environment. Appendix B summarises the syntax of ALGOL 68 by means of a syntax chart due originally to J. M. Watt, J. E. L. Peck and M. Sintzoff.

Throughout the book there are exercises and problems to accompany the chapters. The exercises are intended to test whether the student has a sufficient understanding of the theory in the preceding chapter; sample solutions to these can be found at the end of the book. The problems on the other hand are intended to be programmed and for these no solutions are provided. It is hoped that via both the exercises and problems the student can develop an interest in other branches of computer science. It would also be invaluable if students can be trained not just to answer each question but for every question to ask themselves two other similar questions and answer these.

I should like to take this opportunity of thanking various people for helping me in the preparation of this book. My debt to the authors of the Revised ALGOL 68 Report is obvious. I must also thank J. M. Watt, J. E. L. Peck and M. Sintzoff for their permission to use their syntax chart.

During the writing of the book I had many valuable discussions

on ALGOL 68 especially with Dr. R. B. Hunter but also with Dr. R. Kingslake and other colleagues at Strathclyde. My thanks are due to several people who commented on different parts of the earlier draft of the manuscript. Mr. Ian Walker of the Computer Laboratory, Cambridge University, read the entire manuscript and his comments and our subsequent discussions were most valuable. Other people deserving thanks include Dr. C. Hawksley of the Computer Science Department, Keele University, and Dr. R. Needham of the Computer Laboratory, Cambridge. The typing of the manuscript, and other forms of secretarial work, were performed most willingly and ably by my mother Mrs. M. McGettrick and also by Mrs. M. MacDougall, Miss M. Barron and Miss A. Wisley. On a more personal level I must thank my wife Sheila for her constant support and encouragement throughout the entire period of writing the book.

Finally I am greatly indebted to the Syndics of Cambridge University Press for publishing this book.

Glasgow, April 1976. A. D. McG.

CONTENTS

1

INTRODUCTION TO ALGOL 68

1.1. Preliminaries

The preface to volume one of D. E. Knuth's seven-volume set of books on 'The Art of Computer Programming' begins:

> 'The process of preparing programs for a digital computer is especially attractive because it not only can be economically and scientifically rewarding, it can also be an aesthetic experience much like composing poetry or music. This book is the first of a seven-volume set of books that has been designed to train the reader in the various skills which go into a programmer's craft.'

From this quotation it follows that in an introductory text such as this it will not be possible to cover all aspects of programming.

The word *program* – and *programming* is just the art of writing programs – is rather difficult to define. Roughly speaking, a program is a set of instructions which have usually to be performed or executed by a computer. The instructions which computers actually execute are very simple. They will vary from one make of computer to another but usually include such primitive operations as adding and subtracting two numbers, moving information from one part of the machine to another or reading and printing information. These primitives are called *machine-code instructions*: a large computer may possess as many as 100 or 200 and sometimes more. The set of such instructions for a particular computer together with the method of expressing them forms what is called a *programming language*. Since these instructions are very primitive, such a language is called a *low-level* programming language. It is possible to write programs to solve complicated problems using only machine-code instructions but this can prove a very frustrating introduction to the computer.

Fortunately another class of programming languages has been developed. These so-called *high-level languages* allow the use of a mathematical sort of notation. In a high-level language it might be possible to write

$$a + b \times (c + d)$$

and the computer will evaluate this assuming that *a*, *b*, *c* and *d* had at an earlier part of the program been given some values. If, for instance, *a* had not been given a value the program would be at fault. The sort of instructions one can write in high-level languages are still restricted and it is necessary to learn the different kinds of instructions that are available and the effect these have. The process of carrying out the various instructions in a program will be referred to as the *elaboration* or *execution* of the program.

High-level languages (like natural languages such as English) possess a *syntax* and when writing programs it is necessary to adhere to this syntax. Thus if in writing the above expression a bracket had been omitted then a syntax error would (probably) have been flagged. Syntax errors tend to be caused by such mistakes as omitting brackets, including too many brackets, omitting semi-colons, full-stops, quotation marks and operators. When presenting the computer with a program, therefore, it is preferable that the program should contain no syntax errors, wrong spellings, etc.

The computer will do only as it is asked. It will not deduce the intentions of the careless programmer. If an instruction requests that two numbers be added and the programmer intended that they should have been subtracted, the computer will add them. To counteract this it is advisable to prove if possible that the program does what was intended and to carry out a reasonable number of checks to give confidence in the results.

Earlier it was remarked that the computer executes machine-code instructions. How does it deal with instructions presented in a high-level language? It cannot execute these directly and therefore some form of translation must take place. For each high-level language that a particular computer can handle there is available a (usually fairly large) program called a *compiler* which will take a program written in that language and translate it into machine-code instructions which are then executed by the computer. If the program in the high-level language contains syntax errors (or some other kinds of errors) the compiler should inform the writer of the program of his errors.

The most widely used high-level languages include FORTRAN, COBOL, ALGOL 60, PL/1 and ALGOL 68.

FORTRAN was originally designed *c.* 1955–6 to simplify the writing of programs for numerical calculations. It is one of the most commonly used high-level languages. However, it lacks many modern facilities and is quite unsuitable as a tool for present-day programming.

ALGOL 60, as the name suggests, was designed in 1960 and some

amendments were later added *c.* 1962. Like FORTRAN, ALGOL 60 was designed for numerical calculation. It had a great impact on Computer Science in general.

COBOL is a language whose use is confined almost entirely to the business world. It is used for writing programs to perform tasks such as producing payrolls, factory stock control, etc.

PL/1 and ALGOL 68 are more modern languages and have much wider application than the earlier languages. PL/1 was developed by IBM *c.* 1965 and the original specification of ALGOL 68 appeared in 1968. The revised version of ALGOL 68 appeared in 1974.

The aim of this book is to introduce programming and ALGOL 68. The two topics will be considerably interwoven since the more ALGOL 68 one knows the more sophisticated the programs one can write. ALGOL 68 will therefore have a dual role. It will be used as a high-level language in which to write programs and it will itself be an object of study.

1.2. Remarks on symbolism

In mathematics frequent use is made of symbols to denote constants or variables of different kinds.

(i) The expressions $2\pi r$ and πr^2 give, respectively, the circumference and area of a circle whose radius is r. In these expressions 2 and π represent constants and r is a variable whose value can be any non-negative real number.

(ii) In the expressions $ax + by + c$ and $ax^2 + bx + c$ one might regard a, b and c as real constants and x and y as variables whose values range over the set of real numbers.

(iii) Each of *sin*, *cos* and *tan* is a function which is constant in the same sort of way as, for instance, π is constant. On the other hand a sentence starting "Let f be a function ... " will often indicate that f is a variable whose value can be any function with the stated property. This may be a unique function but there may be more than one function with that property.

Other examples should come readily to mind.

A similar symbolism is used in ALGOL 68. Constants such as *pi*, 14 and *sin* will be used and variables can be introduced. But there are some important differences and it is necessary to be very clear about the values associated with ALGOL 68 constants and variables. Some of these differences arise from the way in which variables are represented in the computer. In order to understand this an extremely simple-minded explanation follows.

The main store of a computer consists of many *locations* (usually

called *words* or *bytes*). The exact number varies from computer to computer but the number is usually expressed in units of K where $1K = 1024 = 2^{10}$. The storage capacity of a modern computer lies between about $4K$ and $1024K$ bytes. These locations are capable of holding different kinds of information e.g. integers, real numbers, characters, etc. and this information can be altered. The locations themselves are accessed by means of *addresses*. The address of a particular location is often thought of as a unique integer in the range 0 to (n-1) where there are n locations in the machine.

When the compiler is translating an ALGOL 68 program into machine-code it will represent a variable by means of an address. Altering the value of a variable amounts to altering the contents of the location with that address. It is extremely important to distinguish between the address of a location and the contents of that location.

1.3. Introduction to ALGOL 68

Some simple examples of ALGOL 68 programs now follow. The examples are trivial but they serve to illustrate certain points. They should not be taken as models of perfect programs. Example 1.3b, for instance, attempts to find the circumference of a circle. It will certainly accomplish this but some instructions should be included to ensure that the radius is not a negative number. Such improvements, together with others of a different nature, could be made but their introduction would cause extra complications which are better avoided for the moment.

Example 1.3a. Write a program to calculate the circumference of a circle whose radius is $e = 2.7182818284$.

```
begin real e = 2.7182818284; real circum;
    circum := 2 × pi × e;
    print (("circumference of a circle of radius e is", circum))
end
```

Clearly this program is extremely limited since it will calculate the circumference of only one particular circle. A more useful program would calculate the circumference of an arbitrary circle (see example 1.3b). A certain generality is desirable in a program.

In order to understand the effect of the various steps in the program above consider successive parts separately.

(i) The (initial) **begin** denotes the start of the program.

(ii) **real** $e = 2.7182818284$ allows e to denote the constant value 2.7182818284. **real** indicates that e represents a real number. The

equals sign is crucial in this section of the program.

(iii) The next action to be executed is that following the semi-colon. For this reason the semi-colon is often referred to as the *go-on symbol*. No mention will be made of the remaining semi-colons in this program. Each semi-colon indicates that the next action to be executed follows that particular semi-colon.

(iv) **real** *circum* causes space to be reserved for a variable and this variable is to be identified as *circum*. The variable takes values which are real numbers.

(v) *circum* := $2 \times pi \times e$ The expression on the right hand side of the *becomes symbol*, i.e. :=, is evaluated. The symbol × denotes multiplication and must be stated explicitly; thus 2*pie* would be illegal. Note that *e* has already been defined and its value thereby known. *pi*, however, is automatically known to the ALGOL 68 compiler. There is no need to define its value.

The effect of the statement is that the value obtained from the evaluation of $2 \times pi \times e$ is given or assigned to the variable *circum*.

(vi) *print* (("*circumference of a circle of radius e is*", *circum*)) The characters between " and " are printed. These are followed by the current value of the variable *circum*. Thus two objects are printed, a string of characters and the current value of a real variable. In the print statement above these two objects are separated by a comma and surrounded by brackets. If a single object, say *circum*, had to be printed then *print* (*circum*) would have been adequate. The output produced by this print statement might look like

circumference of a circle of radius e is $+ 1.7079 \ldots {}_{10} + 1$

where the dots denote the remaining figures after the decimal point and ${}_{10} + 1$ indicates that 1.7079 ... is to be multiplied by 10^1 thus producing 17.079 ... (${}_{10} + 4$ would similarly indicate multiplication by 10^4).

(vii) **end** denotes the end of the program and matches the initial **begin**. Note that there is no go-on symbol between the print statement and the **end.**

Example 1.3b. Write a program to calculate the circumference and area of a circle of arbitrary radius. (The radius is assumed to be non-negative but again no check is included.)

```
begin real r, circum, area;
      read (r);
      circum := 2 × pi × r; area := pi × r × r;
      print (("circumference and area of a circle of radius ", r,
      " are", newline, circum," and", area))
end
```