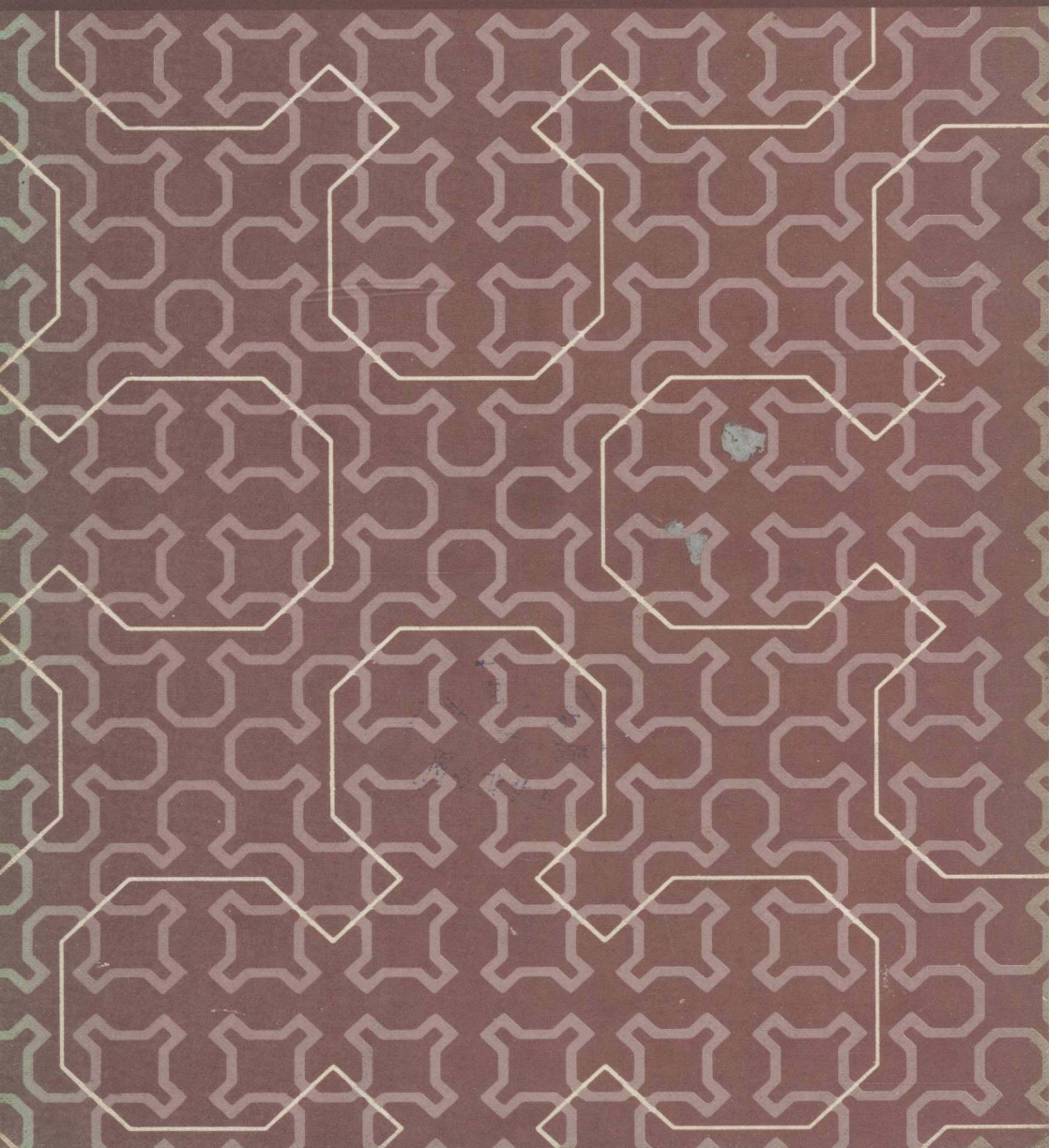


Cambridge Computer Science Texts · 8

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

© Cambridge University Press 1978

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

	Preface	xi
1	Introduction to ALGOL 68	1
1.1	Preliminaries	1
1.2	Remarks on symbolism	3
1.3	Introduction to ALGOL 68	4
1.4	Preparing programs for the computer	7
2	Basic concepts	10
2.1	Modes	10
2.2	Denotations	10
	2.2.1 Integer denotations	10
	2.2.2 Real denotations	11
	2.2.3 Character denotations	12
	2.2.4 Boolean denotations	12
2.3	Identity and variable declarations	12
2.4	The basic arithmetic operators	17
	2.4.1 Addition, subtraction and multiplication	18
	2.4.2 Division	18
	2.4.3 Exponentiation	19
	2.4.4 Monadic operators	20
2.5	Operator priority and bracketing	20
2.6	Comparison and boolean operators	23
2.7	Coercions – dereferencing and widening	25
2.8	Assignations (or assignment statements)	29
2.9	Standard operators	33
	2.9.1 Arithmetical assignment operators	33
	2.9.2 Operators performing mode changes	34
	2.9.3 Other operators	35
2.10	Layout, comments and pragmatic remarks	35
2.11	Simple transput	37
	2.11.1 The read statement	38
	2.11.2 The print or write statement	40
	2.11.3 The conversion routines	43
	Exercises	47
	Programming problems	51
3	Clauses	53
3.1	Environments	53

3.2	Unitary clauses	55
3.3	Serial clauses	56
3.3.1	Value associated with a serial clause	57
3.3.2	Voiding	57
3.4	Closed clauses	58
3.4.1	Value associated with a closed clause	58
3.4.2	Closed clauses and coercions	59
3.5	Ranges and reaches	60
3.5.1	'Equivalence' in ALGOL 68	65
3.5.2	Defining and applied occurrences	66
3.5.3	Results delivered by closed clauses	67
3.6	Collateral phrases	67
3.6.1	Void-collateral-clauses	68
3.6.2	Collateral declarations	70
3.7	Loop clauses	70
3.7.1	Illustrative examples	70
3.7.2	General form of loop clauses	72
3.7.3	Use of skip	74
3.7.4	Ranges associated with loop clauses	75
3.8	Choice clauses	76
3.8.1	Conditional clauses	76
3.8.2	Case clauses	79
3.8.3	Ranges associated with choice clauses	81
3.8.4	Balancing	82
3.9	Jumps	84
3.9.1	Labelling and jumping	84
3.9.2	Restrictions on jumps	86
3.9.3	Special cases of jumps	88
3.10	Programs in ALGOL 68	90
3.10.1	Enclosed clauses	90
3.10.2	General remarks about programs	91
3.10.3	Structured programming	93
3.10.4	Miscellaneous examples on programming	94
	Exercises	98
	Programming problems	100
4	Multiple values and simple structures	103
4.1	Multiple values	103
4.1.1	Declarations and subscripting	104
4.1.2	Modes associated with multiple values	106
4.1.3	Multiple values of several dimensions	107
4.1.4	Slicing	108
4.1.5	Assignations involving multiple values	111
4.1.6	Row displays	113
4.1.7	Character manipulation	115
4.1.8	Simple transput involving multiple values	117
4.1.9	Rowing	118

4.1.10	Variable declarations and equivalent identity declarations revisited	119
4.1.11	Programming using multiple values	122
4.2	Simple structures	124
4.2.1	Declarations and selection	124
4.2.2	Modes associated with structures	126
4.2.3	Building more complex structures	127
4.2.4	Multiple selection	129
4.2.5	Assignations involving structures	129
4.2.6	Structure displays	130
4.2.7	Transput involving structures	131
4.2.8	Programming using simple structures	132
4.3	Simple mode declarations	134
4.3.1	Examples of mode declarations	135
4.3.2	row-of-row-of-modes	136
4.3.3	Rowing revisited	137
	Exercises	138
	Programming problems	140
5	Procedures and operators	143
5.1	Procedures	144
5.1.1	The standard procedures	144
5.1.2	Modes associated with procedures	147
5.1.3	Routine texts	149
5.1.4	The ALGOL 68 calling mechanism	154
5.1.5	Recursive procedures	155
5.1.6	Deproceduring	158
5.1.7	Deproceduring and voiding	160
5.2	Operators	161
5.2.1	Extending the definition of operators	162
5.2.2	Introducing new operators	164
5.2.3	The uniqueness condition for operators	168
5.2.4	Identification of operators	171
5.3	The use of procedures and operators in programming	172
	Exercises	173
	Programming problems	176
6	More standard modes	179
6.1	Complex numbers	179
6.2	The modes bytes and bits	182
6.2.1	Bytes	183
6.2.2	Bits	184
6.3	Multiple length facilities	187
6.3.1	Environment enquiries associated with multiple length facilities	188
6.3.2	Multiple length denotations and related topics	189

6.3.3	Extensions of standard operators	190
6.3.4	Standard constants and functions	192
6.3.5	Transput	193
	Programming problems	193
7	Advanced features associated with modes	195
7.1	Flexible names	195
7.1.1	Declaring flexible names	196
7.1.2	Deflexing	197
7.1.3	Transient names	199
7.1.4	flex and declarers	201
7.1.5	The standard mode string	202
7.2	United modes	203
7.2.1	Uniting	204
7.2.2	Component modes	206
7.2.3	Conformity clauses	208
7.2.4	Overheads in using united modes	211
7.3	Orthogonality – modes and constructions	211
7.3.1	Modes	212
7.3.2	Actual, formal and virtual declarers	212
7.3.3	Constructions	214
7.3.4	Pointers	215
7.3.5	Casts	216
7.3.6	Identity relations	216
7.4	Orthogonality – coercions	218
7.4.1	NONPROC modes revisited	218
7.4.2	Weak dereferencing	219
7.4.3	Review of coercions and syntactic positions	221
7.5	Orthogonality – scope	222
7.5.1	Scope revisited	222
7.5.2	Local generators revisited	224
7.5.3	Global or heap generators	226
7.5.4	Scope of routines	228
7.6	Recursive modes	229
7.6.1	Well-formed modes	231
7.6.2	Equivalent modes	233
7.7	Programming examples	234
7.7.1	Using flexible names	234
7.7.2	Using recursive modes	235
7.7.3	Using united modes	237
	Exercises	238
	Programming problems	242
8	Parallel processing	247
8.1	Introductory remarks	247
8.2	The mutual exclusion and message passing problems	248

8.3	Parallel clauses and semaphores	250
8.4	Solution to mutual exclusion problem	251
8.5	Solution to message passing problem	252
8.6	The dining philosophers problem	256
8.7	References on parallel processing	258
	Programming problems	259
9	Transput	262
9.1	Books, channels and files	263
9.1.1	Books	264
9.1.2	Channels	266
9.1.3	Files	269
9.2	Formatless transput	272
9.2.1	Straightening	272
9.2.2	Linking books to files	273
9.2.3	The <i>get</i> and <i>put</i> routines	276
9.2.4	<i>space</i> , <i>newline</i> , etc. revisited	277
9.2.5	Association	277
9.3	Accessing and altering files	279
9.3.1	File enquiries	279
9.3.2	The event routines	280
9.3.3	Altering identification of books, terminator strings and character conversion codes	284
9.3.4	Setting and resetting files	285
9.3.5	Manipulating files	288
9.4	Formatted transput	290
9.4.1	Format texts	292
9.4.2	Pictures	293
9.4.3	Insertions	294
9.4.4	Patterns, moulds and frames	295
9.4.5	Use of the formatted transput routines	296
9.4.6	The event routines revisited	297
9.4.7	Sign moulds	299
9.4.8	Special frames	300
9.4.9	Patterns	302
9.4.10	Pragmatic remarks and comments in format texts	309
9.4.11	Controlling transput using formats	309
9.4.12	Manipulating formats	311
9.5	Binary transput	312
	Exercises	315
	Programming problems	318
	Answers to exercises	321
	Appendix A – the standard environment	330
	Appendix B – the syntax chart	339
	Index	343

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 counter-act 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 :=  $2 \times \pi \times e$ ;  
      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 \times denotes multiplication and must be stated explicitly; thus $2\pi e$ would be illegal. Note that e has already been defined and its value thereby known. π , 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 ... $_{10} + 1$

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 :=  $2 \times \pi \times r$ ; area :=  $\pi \times r \times r$ ;  
  print ("circumference and area of a circle of radius ", r,  
    " are", newline, circum, " and", area))  
end
```