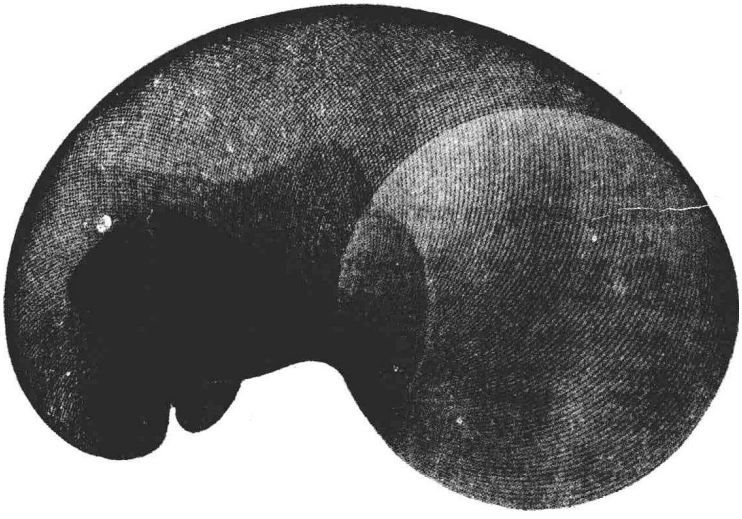
Special English Peter Strevens General Editor

Philip Bedford Robinson

Computer Programming

Philip Robinson

Computer Programming



Cassell
London

Grateful acknowledgement for permission to use the photographs and diagrams reproduced in this book is made to the following (numbers refer to the pages on which the pictures appear):

British Overseas Airways Corporation (45); IBM (UK) Limited (viii, 10, 40, 51, 57, 67); International Computers Limited (iii, 2, 7, 13, 18, 20, 33, 35, 38, 39, 41, 52, 60, 64, 69, 72).

Cover photograph: International Computers Limited

The pattern on the title page was produced by a computer.

Cassell Ltd.
10 Greycoat Place, London SW1P 1SB

First printing 1972
Second printing 1979
Third printing 1981
Fourth printing 1982

ISBN 0 304 30412 3

Printed and bound in Great Britain at
The Camelot Press Ltd, Southampton

PREFACE

This Special English series introduces titles on a wide range of technical subjects that will be of interest to students of English as a second language. Each volume illustrates the special English of a particular trade or profession in both its spoken and written forms. It is not possible, of course, for books of this size to cover the subject matter exhaustively, so the authors have concentrated on those topics and activities that should have the widest appeal. The conversations which are the basis of each chapter or unit are deliberately written in the colloquial and idiomatic speech used by technicians and specialists as they go about their everyday activities.

It must be emphasized that these books are *not* intended to teach the subject matter itself, although the technical content is accurate in every respect. Nor are they intended to teach the introductory stages of English. It is assumed that the reader is already familiar in his own language with the subject matter of the book, and has a good grounding in the basic grammatical patterns and vocabulary of English. He will use these books to improve his knowledge of English within the framework of a technical vocabulary that is of interest to him either privately or professionally.

The authors in this series each have their individual approach, but all the volumes are organized in the same general way. Typically, each book is based on a series of situational dialogues, followed by narrative passages for reading comprehension. Exercises give the student practice in handling some of the useful and more difficult patterns, as well as lexical items, that occur in each unit. Tape recordings, of the dialogues and selected exercises, may be used either in the language laboratory or for private study. Each volume is provided with a glossary of technical terms, with i.p.a. equivalents as used in the Daniel Jones Pronouncing Dictionary.

PETER STREVENS
General Editor

INTRODUCTION

This book is a sequel to *Special English: Computers*, which discussed the concepts involved in both hardware and software, as well as the work of a variety of computer personnel.

Computer Programming concentrates on the work of a key figure in the new world of computers, the Programmer. It explains what he does, the techniques he uses, and the different fields in which he may specialize. But like the other books in the series it is primarily designed to teach English in the context of a particular occupation. Each unit includes a Dialogue, in which programmers talk to each other using the register of their profession; a Reading and Comprehension passage; and Exercises, for structural practice and comprehension.

At the end of the book there are Keys to the Exercises, and a Glossary of technical terms (which are asterisked on their first occurrence in the text). The International Phonetic Alphabet is used as a guide to pronunciation. Colloquial expressions are footnoted.

The book is not a substitute for a course in computer programming, but it will enable a student whose mother tongue is not English to take such a course with confidence.

The tape recording that accompanies the book may be used by the teacher in the classroom or the language laboratory. For the student working alone, it will provide a model for pronunciation as well as a means of taking dictation for practice in spelling. The exercises have pauses for student response, but there are no pauses in the dialogue. This has been done on purpose to provide the maximum amount of recorded material. Most tape recorders are now equipped with a pause button which enables the listener to stop the tape after each sentence and repeat it aloud before proceeding to the next one. If pauses are required for language laboratory work, a copy may be made and the pauses inserted of a length to suit the requirements of the students.

store occupancy ['stɔ: 'ɔkjupənsi] The number of **store locations** occupied by a program while it is being run.

subfile ['sʌbfail] A file contained within the structure of another file on a storage medium.

subroutine ['sʌbru:ti:n] A set of instructions designed to perform a specific task and capable of being obeyed over and over again by a program, or by more than one program. It is therefore ideally written as a self-contained **module**.

subscript ['sʌbskript] The numeric **key** which identifies a particular element in a set or table.

subset ['sʌbset] A set which is part of a larger set.

symbol state table ['simbl ˌsteit ˌteibl] A table whose elements specify that different **subroutines** are to be performed according to the values of successive input **parameters**. The parameters may be, for example, the characters of a word in a source program that is being analysed by a compiler.

syntactic [sin'tæktik] Concerning the rules governing the relations of words in language structures.

systems analyst ['sistəmz ˌænəlist] One who investigates the data processing of an organization to define how it may best be performed by a computer, and writes the **specifications** from which the computer programs are written.

systems programmer ['sistəmz ˌprougræmə] A programmer who writes generalized routines designed to be used in all the installations of a particular computer, such as compilers, utility routines, sort programs, etc.

table ['teibl] A set of data organized so that each element may be uniquely identified by a **key** held in the element, or by the position of the element in the table.

tabulators ['tæbjuleitəz] Data processing machines which antedated the computer, and which could read packs of punched cards, tallying fields and printing out totals and subtotals. They could be programmed by means of **plugboards**.

tallying ['tæliiŋ] Counting, or keeping count.

test data ['test deitə] Data often artificially created, to test a program under development.

trace routine ['treis ru:ˌti:n] A diagnostic aid which monitors the progress of a program by printing out the contents of specified areas, signalling all transfers of control and so on.

transaction processing [træn'sækʃən ˌprousesiŋ] The processing of information piecemeal, as it becomes available or at specified intervals, instead of waiting until it is batched up with other similar information.

tree [tri:] A set of data organized as a hierarchy.

truncation [trʌŋ'keiʃən] The deletion of digits from one end or another of a number, e.g. because it is too big to hold in the **storage location** allocated to it.

utility routine [ju:'tili ru:ˌti:n] A standard library program of frequent use in all computer systems, such as a program for printing out the contents of a magnetic tape.

validate ['vælideit] Check for correctness, e.g. of **input data**. *Noun:* **validation**.

validity check [və'liditi ˌtʃək] A check that data conforms to certain rules or is within certain limits.

vector ['vektə] A one-dimensional **table**.

virtual store ['və:tʃuəl ˌstɔ:] A concept which allows the programmer to consider that he has available to his program more main store than the physical **core store** of the computer for which he is programming.

write ring ['rait ˌriŋ] A physical device attached to a magnetic tape reel to allow, or prevent, writing to that reel.

UNIT 1

THE JOB OF THE COMPUTER PROGRAMMER

Dialogue

Peter Bracknell is joining a computer manufacturer as a junior programmer.

Manager: Glad to have you with us, Peter. Have you done any programming before?

Peter: I'm afraid not. In fact I don't really know what programming involves. I took the aptitude test, of course, but all that told me was that you wanted to find out if I was any good at solving logical and semantic problems.

Manager: Well, evidently you are, so that's half the battle.[1]

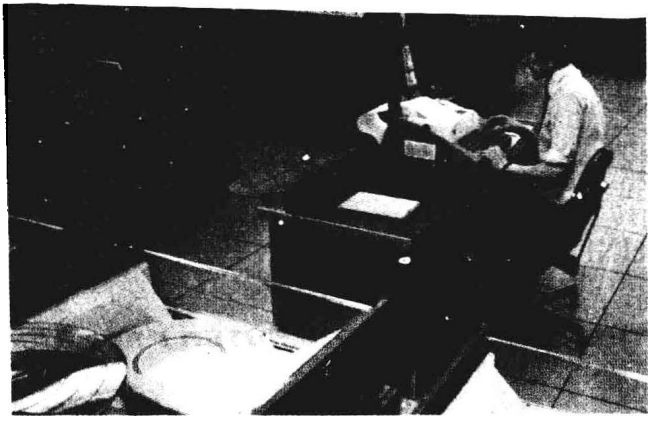
A computer programmer is first and foremost[2] an interpreter. He's given a problem described in a natural language such as English, and he has to break it down into logical steps and then translate the steps into a language understood by the computer.

Peter: What sort of problems do programmers have to deal with?

Manager: Basically, three types. Firstly, scientific: things like weather forecasting, statistical analysis, integration—anything related to sciences such as physics, biology, mathematics, astronomy, or the technologies used in industry.

Many scientific programmers work in universities or research establishments. Some of them are scientists or technologists first and programmers second. They only want to know enough programming to solve their own particular problems.

[1] half the battle: half the problem has been dealt with successfully
[2] first and foremost: mainly, predominantly



Peter: But it's different with commercial programmers, isn't it? I mean, they often go straight into programming from university, rather than working as accountants or bank clerks first.

Manager: A lot of them do. They may write programs to handle invoicing, or share registration, or stock control, but they only learn about these things as they go along. The *specifications of their problems are written by *systems analysts.

Peter: Which type of work shall I be doing?

Manager: Neither! We're working for a computer manufacturer, not a user, so we're called *systems programmers. We write *software that acts as a *buffer between scientific and commercial programs, and the machine. Things like *compilers and *assemblers, *executives and *operating systems—not to mention *utility routines to do such things as sorting and printing out the contents of magnetic tapes and *discs.

Peter: Because every user has to sort his files and print out his magnetic *media, so there's no point in each one writing his own program to do them?

Manager: That's right. And when we sell a computer we supply all this systems software as part of the package deal—though nowadays some manufacturers are selling the software separately.

But whatever type of programs they write, scientific, commercial, or systems software, all programmers have three main objectives.

They must write programs that work. They must write them on time. And they must leave them fully *documented so that they can be easily taken over, maintained and amended by other programmers.

EXERCISE 1: STRUCTURAL PRACTICE

Notice this structure from the conversation:

(No), *We're* work*ing* for a computer manufacturer, *not* a user.

Use this structure to respond to the following questions:

Example: Are we working for a user?
Prompt: *computer manufacturer*
Response: No, we're working for a computer manufacturer, not a user.

Now you do it.

1. Are we working for a user?	***computer manufacturer***
2. Are we testing specifications?	***programs***
3. Are we writing assemblers?	***compilers***
4. Are we solving scientific problems?	***commercial problems***
5. Are we selling the hardware?	***software***
6. Are we running the installation?	***marketing side***
7. Are we concerned with programming?	***systems analysis***
8. Are we printing out magnetic tapes?	***discs***

EXERCISE 2: PROGRESSIVE SUBSTITUTION DRILL

Statement: What sort of problems do programmers have to deal with?
Prompt: *solve*
Response: What sort of problems do programmers have to ***solve***?

Now you do it.

Statement: What sort of problems do programmers have to deal with?

Prompts:

1. solve	5. jobs
2. type	6. operators
3. managers	7. techniques
4. handle	8. learn

EXERCISE 3: FURTHER STRUCTURAL PRACTICE

Change the following statements into questions:

1. The specifications are written by systems analysts.
2. Every user has to sort his files.
3. We supply all this systems software.
4. We're working for a computer manufacturer.
5. He has to break it down into logical stages.
6. Some of them are scientists and technologists.
7. They go straight into programming from university.
8. They can be easily taken over by other programmers.

Reading and Comprehension

A program is a sequence of instructions that must be obeyed to achieve a given result. A knitting pattern or a carpet design is a program, and so are the directions one is given to arrive at some destination: "Take the second turning on the left, bear right at the post office, continue to the first roundabout and then take the third exit . . ." If your car won't start, the series of tests you make is a program, and a mathematical formula such as $x = (a^2 + b^2)/2ab$ is another type, instructing you to perform certain operations on the variables a and b to arrive at a value for x.

A computer program is analogous to all these, and differs from them only in being written in an artificial language, sometimes similar to algebra or a natural language such as English, but restricted to the types of operation that a computer can perform. A computer programmer has to learn such languages. He need not be a mathematician, but he has to have the ability to think logically, and he will need training in special programming techniques.

EXERCISE 4: QUESTIONS ON THE DIALOGUE AND READING PASSAGE

1. What is a computer programmer, first and foremost?
2. What are the three different types of program?
3. Name some examples of systems software.
4. Is a knitting pattern a program?

5. Give an example of a natural language.
6. Must a programmer be a mathematician?
7. What sort of tasks are performed by utility routines?
8. Name some examples of commercial programs.

EXERCISE 5

Complete the following sentences, using the appropriate words from the list below:

a. commercial
b. carpet
c. specifications
d. main
e. logical
f. separately
g. sort
h. establishments

1. He has to break it down into ——— steps.
2. Many scientific programmers work in research ——— .
3. It's different with ——— programmers, isn't it?
4. The ——— of their problems are written by systems analysts.
5. Every user has to ——— his files.
6. Some manufacturers are selling the software ——— .
7. All programmers have three ——— objectives.
8. A ——— design is a program.

UNIT 2

THE PROGRAMMER'S TOOLS

Dialogue

Peter chats with his section leader, Geoff.

Peter: I've just been round the machine room and seen all the *hardware—the *central processor and *input-output units. Thought I'd better get to know the tools we use—though they're a bit more expensive than hammers and chisels!

Geoff: Yes, but the hardware is chiefly used by the operators and engineers. We seldom go near it. What you've got to learn to use are our software tools.

Peter: You mean program specifications and *flowcharts?

Geoff: And more specialized things like computer languages. There's a whole spectrum of them, ranging from "high-level" to "low-level", each designed for a special purpose. Languages such as *COBOL, *FORTRAN, *ALGOL, *JOSS, *APT, *BASIC, *PL/1 and dozens of others. But don't get alarmed; you don't have to learn all of them. And their vocabularies only contain a hundred or so words. Not like English, with nearly half a million!

Peter: What do you mean by "high-level" and "low-level"?

Geoff: "High level" languages are *problem-oriented, similar to natural languages or algebra. "Low-level" are machine-oriented, closer to the language which the machine understands.

Peter: The lowest level being *machine code?

Geoff: Yes. As you know, the *central processor contains thousands of tiny circuits, each of which can be in one of two states, either "on" or "off" like a switch. Now, if you combine these circuits in groups of, say, six, you can get sixty-four possible combinations of switch settings for each group. Each combination can activate a different function of the machine.

Peter: I see. You mean, one might start up the card reader?

Geoff: Yes, and another might add the contents of a *store location into a fast *register. In other words, each combination is an instruction to the machine, and it's perfectly possible to feed them directly into the machine from a *console, just as you set the switches on a control panel.

In any machine code—there's a different set for each different computer—there are usually about fifty to a hundred and fifty different instructions.

Peter: But you said there could only be sixty-four different combinations.

Geoff: That's if you combine the circuits in groups of six. But you can also combine them in groups of eight, for example.

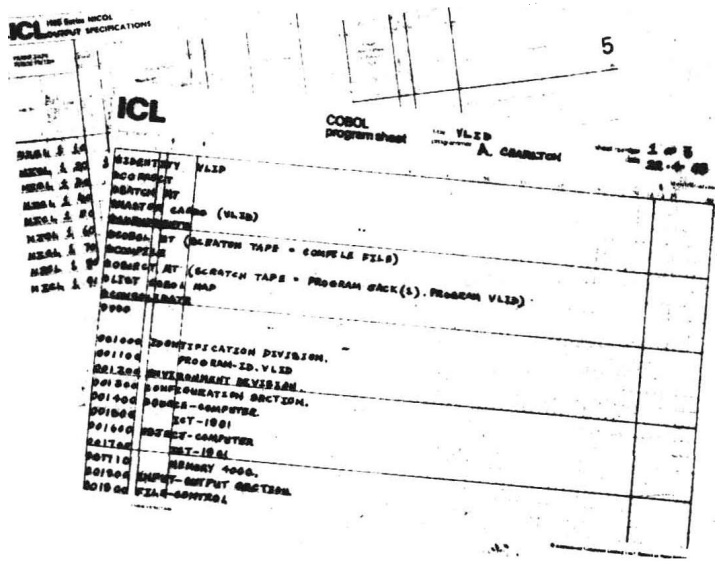
Peter: So to tell the machine to read a magnetic tape you'd have to type in a binary pattern—101011, say—at the console?

Geoff: That's what programmers did in the beginning. But nowadays they can *punch a sequence of instructions on cards, and then type an instruction to the machine to read the cards and obey the instructions on them.

Peter: It must be a bit tedious, though, to have to remember the binary patterns for all the possible instructions.

Geoff: It is. That's why *assembly languages were devised. Instead of punching 101011, you simply punch RD, the *mnemonic for "Read".

COBAL program sheets



Peter: But how can the machine understand these mnemonics?

Geoff: It can't. You have to have a special *assembler program to translate the mnemonics into machine code. That's an example of the sort of programs we write, a program that translates another program from source form into machine code.

Peter: Is an assembly language "high-level", then?

Geoff: Sorry, no. It's still low-level because it's one-for-one: one mnemonic such as LDX is translated into one machine code instruction meaning "Load into Accumulator" An instruction in a high-level language, on the other hand, often translates into several machine code instructions, sometimes as many as forty.

Peter: So I'll be learning some high-level and low-level languages on my course?

Geoff: Yes, but since your course doesn't start for some weeks you'll be learning some basic programming techniques to start with—techniques you'll use whatever language you're writing in.

EXERCISE 1: STRUCTURAL PRACTICE

Notice this structure from the conversation:

(No,) the hardware *is* us*ed by* the operators.

Use this structure to respond to the following questions:

Example: Do we use the hardware?
Prompt: *the operators*
Response: No, the hardware is used by the operators.

Now you do it.

1. Do we use the hardware?	*the operators*
2. Do we write the specifications?	*systems analysts*
3. Do we punch the cards?	*DP staff*
4. Do we make those decisions?	*committees*
5. Do we invent the instructions?	*language designers*
6. Do we design the software?	*senior programmers*
7. Do we learn these languages?	*commercial users*
8. Do we maintain the peripherals?	*the engineers*

EXERCISE 2: PROGRESSIVE SUBSTITUTION DRILL

Statement: You don't have to learn all of them.
Prompt: *We*
Response: *We* don't have to learn all of them.

Now you do it.

Statement: You don't have to learn all of them.
Prompts:

1. We
2. the languages
3. know
4. the peripherals
5. use
6. Don't we
7. operate
8. machines

EXERCISE 3: FURTHER STRUCTURAL PRACTICE

Change the following sentences into the passive.

Example: You combine the circuits in groups of six.
Response: The circuits *are combined* in groups of six.

Now you do it.

1. You combine the circuits in groups of six.
2. You'd have to type in a binary pattern.
3. They can punch a sequence of instructions on cards.
4. That's what programmers did in the beginning.
5. How can the machine understand these mnemonics?
6. What do "high-level" and "low-level" mean?
7. The central processor contains thousands of tiny circuits.
8. One instruction might start up the card reader.

Reading and Comprehension

Low-level languages are "machine-oriented" because each instruction is a mnemonic representing a single machine code binary instruction. In a high-level language such as COBOL, however (COBOL is an *acronym for Common Business Oriented Language), an instruction such as ADD A TO B may generate three machine code instructions:

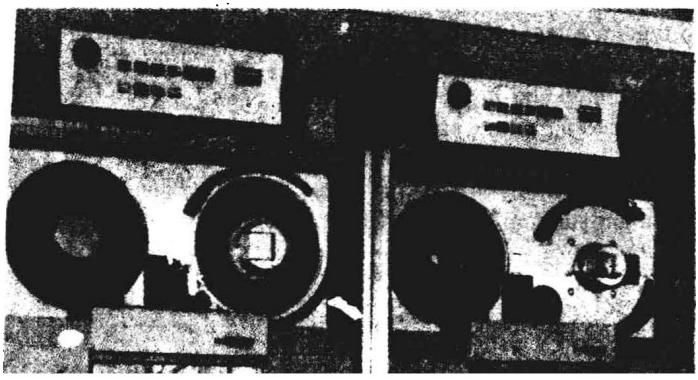
Load A into Fast Register
Add B into Fast Register
Store contents of Fast Register in B

The COBOL instruction OPEN, when translated by a compiler, may generate even more machine code instructions to perform such functions as:

activate a particular tape deck;
read the first block on the tape;
check whether the name in the first block is the name of the file to be read;
perform other checks on the "tape label";
set a switch defining the file as "open".

A program written in a high-level language is called a "source program"; the machine code version into which it is translated is an "object program".

Writing in a high-level language is easier and quicker for a programmer than using machine code, but there are two other major advantages. It is *compatible*, i.e., his program can be run on *any* machine, provided there is a compiler to translate it into that machine's code. Then, addresses in his program can be given *names*: all addresses can be relative to the addresses of the names, and this facilitates the transference of his program from one machine to another.



Magnetic tape units

EXERCISE 4: QUESTIONS ON THE DIALOGUE AND READING PASSAGE

1. Name some computer languages.
2. Are low-level languages problem-oriented?
3. Which is the lowest-level language?
4. What does "one-for-one" mean?
5. What is a mnemonic?
6. What is the translated version of a source program called?
7. Name some advantages of writing in high-level languages.
8. Approximately how many instructions are there in any machine code?

EXERCISE 5

Use the following words and phrases in sentences of your own to show that you understand their meaning and use:

1. compatible
2. assembler program
3. software
4. vocabulary
5. instruction
6. binary
7. techniques
8. punch