ISBN 0 471 01650 0

Programs and Machines

An Introduction to the Theory of Computation

Richard Bird
Department of Computer Science,
University of Reading

A Wiley–Interscience Publication

JOHN WILEY & SONS
London · New York · Sydney · Toronto

Library of Congress Cataloging in Publication Data:

Bird, Richard, fl. 1969–
  Programs and machines.

  (Wiley series in computing)
  'A Wiley–Interscience publication.'
  Bibliography: p.
  Includes index.
  1. Electronic digital computers—Programming.
I. Title.
QA76.6.B57   001.6′42   75–38893

ISBN 0 471 01650 0

Photoset in Great Britain by
Tecnical Filmsetters Europe Limited,
76 Great Bridgewater Street, Manchester M1 5JY
and printed by The Pitman Press, Bath

Programs and Machines

To my wife Norma

and my parents, Kay and Jack

Preface

This book is intended primarily for programmers and computer science students, either at the undergraduate or first-year graduate level, who desire a self-contained introduction to the theory of computation. Throughout, I have attempted to develop the subject from programming concepts, and not as an abstract mathematical theory. I have assumed that the reader possesses a basic knowledge of computers and programming languages (hopefully, but not essentially, including at least one of the Algol variety) and have built upon this knowledge in the selection and discussion of topics. Naturally there are results of a mathematical nature, but I have tried to present the proofs as simply as possible, often resorting to proof by example when a rigorous argument would involve too many tedious details. Mathematically, the book requires no formal background apart from some elementary algebra and a familiarity with modern mathematical notation.

The theory of computation has undergone many shifts of emphasis in the last fifteen years. Generally speaking, the trend has been away from the study of formal computing devices, and towards a basic understanding of program structures. This trend is reflected in the present material. Starting off with a discussion of three basic types of program (Chapter 1), the book covers the equivalence of programs (Chapter 2), the limitations of programming (Chapters 3 and 4), the correctness of programs (Chapter 5), and the theory and use of recursion (Chapters 6, 7, and 8). In particular, as the experienced reader will notice, no mention is made of the theory of automata, finite or otherwise. One simple reason for this apparently major omission is that automata theory has already received comprehensive treatment in several excellent existing texts. A second reason is that I have followed the recommendations of Dana Scott* and drawn a fundamental distinction between the concepts of program and machine, a distinction which underlies the whole book. Given this approach, it is inappropriate to include material from automata theory in which this distinction is not drawn. In any case, many automata-theoretic results can be reformulated as results about programs; for instance, Chapter 2 deals essentially with the state equivalence of finite automata in a different guise.

* Scott, D. (1967) Some definitional suggestions for automata theory. *Journal of Computer and System Sciences* 1 (2), 187–212.

Rather than over-burden the text with extensive attributions, I have included bibliographic comments and references at the end of the book. Above all, I have profited from the papers of John McCarthy and Dana Scott on the foundations of the subject.

This book developed from the courses I have given at the University of London Institute of Computer Science, the University of British Columbia, and the University of Reading. It is a pleasure to record my gratitude to Michael Bell, David Cooper, Peter Landin, Ray Reiter, Richard Rosenberg and David Till whose comments and suggestions have helped me greatly. Finally, special thanks go to the ever patient Margaret Lambden who typed the manuscript.

University of Reading RICHARD BIRD
March 1975

Contents

Chapter 1
Programs and Machines

We must begin our investigation into the nature of computation by defining exactly what we mean by a program and a machine. Rather than attempt to describe any existing programming language or computing machine, of which there is a great abundance and variety, we shall characterize their essential features in simple mathematical models. Throughout, we shall treat programs and machines as distinct but complementary entities, which come together, on an equal footing, to define computations. As well as being natural, this division enables us to consider particular aspects of computation, such as the properties of a certain type of program structure or the capabilities of a certain machine, in isolation from the rest of the computational process, and so avoid unnecessary detail.

1.1 Programs

Speaking generally, a program can be regarded as a structured set of instructions which enables a machine—human or mechanical—to successively apply certain basic operations and tests in a strictly deterministic fashion to given initial data, until the data has been transformed into some desirable form. For the moment we do not wish to be too concerned with the precise nature of the operations and tests which constitute instructions, so we shall just give them names. We suppose the existence of two sets of *identifiers*

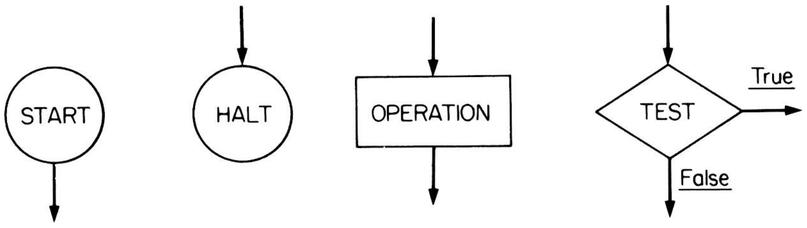
operation identifiers	$F,G,H,\ldots$
test identifiers	$T,U,V,\ldots$

Thus F denotes an operation of some sort, and T denotes a test. A test is just an operation which produces one of two possible truth-values, *true* or *false.*

In order that these operations and tests can be carried out in a prescribed manner, the program has to provide a control structure for its constituent instructions. This structure determines the flow of the computation. Among the many examples of such structures in present day programming languages, we select three representative types for study. These structures lead directly to the definition of three types of program: flowcharts, **while** programs, and procedure definitions. We consider the most important one first.

(a) Flowchart programs

Informally, a flowchart program is a geometric diagram made up, according to certain rules, of the following types of components:



A simple example of a flowchart program is given in Figure 1.

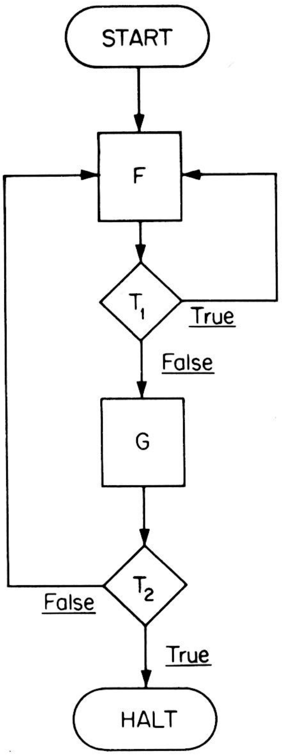


Figure 1. A flowchart

With the judicious use of labels and jumps, flowchart programs can also be specified by sets of labelled instructions. Thus Figure 1 can be equally well described by the following set of instructions:

```
1: do F then goto 2
2: if T_1 then goto 1 else goto 3
3: do G then goto 4
4: if T_2 then goto 5 else goto 1.
```

Two assumptions have been made in translating the flowchart into the above form. Firstly, the computation must begin at label 1, and secondly, the computation should terminate successfully when an attempt is made to jump to a non-existent instruction. This indeed will be the case when we come to define computations with sets of labelled instructions.

The formal definition of the class of flowchart programs is best given by using this idea of a labelled instruction. We suppose that L is some standard set of *label identifiers* (usually, the set of arabic numerals).

A *labelled instruction* i is a string of symbols of one of the two forms:

l: **do** F **then goto** l'

l: **if** T **then goto** l' **else goto** l'',

where F is an operator identifier, T is a test identifier, and l, l', and l'' are label identifiers in L. The label $\lambda(i)$ of instruction i is the label identifier to the left of the colon sign in i.

A *flowchart program* (or more shortly, a *flowchart*) P is a finite set of labelled instructions with the property:

$$\text{for all } i, j \text{ in } P, \quad \text{if } \lambda(i) = \lambda(j), \quad \text{then } i = j.$$

In other words, P is a set of instructions no two of which can have the same label. In addition, P also specifies a particular label identifier in L. This label is referred to as the *initial label* of P.

A *terminal label* of P is a label l appearing in some instruction of P such that for no i in P do we have $l = \lambda(i)$. Thus, in the flowchart of Figure 1 represented as a set of labelled instructions, label 1 is the initial label, and label 5 is the unique terminal label. By introducing these special labels we can avoid having to specify explicit start and halt instructions for each program. Notice that the definition of a flowchart does not absolutely require either the presence of terminal labels, or the presence of an instruction with the initial label. Indeed, it is possible that P contains no instructions whatsoever. The sort of computations such programs carry out will be described in the appropriate place.

Arguably, the sequencing mechanism given by labelled instructions is the most fundamental type of control structure. It is the basic structure utilized by most assembly and machine languages, and is incorporated in some way or other in most 'high-level' programming languages. On the other hand, the next control structure has made its appearance in programming languages only comparatively recently.

(b) While programs

While programs are based on three sequencing mechanisms which can be found in a number of present day high-level languages (e.g. ALGOL 68, PASCAL). They are described as follows:

(i) *composition.* Suppose that P and Q are two programs. Their composition, which we write as $P; Q$, denotes a further program whose effect is to execute P and Q in the order: P followed by Q. More generally, since composition is clearly associative, we can write $P_1;\ P_2; \ldots;\ P_n$ for the program which executes $P_1, P_2, \ldots, P_n$ from left to right.

(ii) *conditional statements.* Suppose that P and Q are two programs. The statement

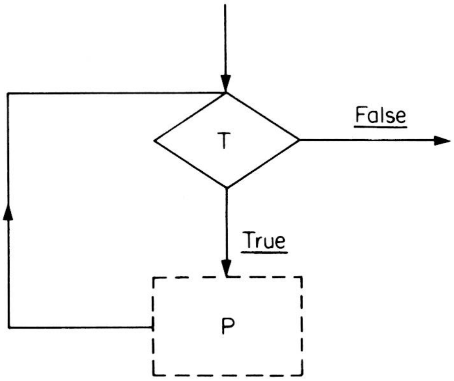
(if T **then** P **else** Q**)**

denotes the program which executes P if T is *true*, or executes Q if T is *false*.

(iii) **while** *statements.* Suppose P is a program. The statement

while T **do** P

denotes the program which repeatedly tests T and performs P until the result of testing T yields the value *false* (if it ever does), in which case the iteration terminates. The same sequence of operations is carried out by the following flowchart.



Since termination can only be caused by returning a value *false* for the test identifier, it is convenient to introduce the further statement

until T **do** P,

whose effect is to repeatedly test T and perform P until the value of T is *true*. The statement **until** T **do** P is therefore equivalent to **while** *not-T* **do** P; we prefer, however, not to introduce negations of basic test identifiers.

In order to define the class of **while** programs formally, we introduce the idea of the null program I, which corresponds to the dummy operation 'do nothing'. It is clear that I acts as an identity element under composition, i.e. $I; P = P; I = P$ for any program P. The class of **while** programs is defined recursively according to the following rules:

1. Each operation identifier standing by itself is a **while** program; so, conventionally, is the null program I.
2. If W and V are **while** programs, then so are each of the following:
   (2a) V; W,
   (2b) **(if** T **then** V **else** W),
   (2c) **while** T **do** (V)
   (2d) **until** T **do** (V)

   where in (2b)–(2d), T is an arbitrary test identifier.

Notice that the pairs of brackets which appear in the constructions (2b)–(2d) are necessary if we wish to be able to unambiguously analyse a **while** program into its constituent parts. Otherwise, for example, the program

while T **do** V; W

would admit two distinct interpretations, corresponding to

while T **do** (V); W and **while** T **do** (V; W).*

It may appear that nothing much has been gained by considering **while** programs. It is easy to see, in an intuitive way, that **while** programs can be translated into flowcharts without any loss of information. This remark will be made more precise later. There are two basic reasons why **while** programs are still worth considering. Firstly, the converse of the above remark is false, so that the class of computations that can be carried out by **while** programs is a proper

* Occasionally, especially in Chapter 5 onwards, we shall use the Algol symbols **begin** and **end** instead of parentheses to delimit the extent of the **while** statement. Thus

while T **do begin** V; W **end**

is regarded as an alternative way of writing

while T **do** (V; W).

When V consists of a single operation identifier F we shall sometimes omit parentheses, i.e. we write

while T **do** F

instead of **while** T **do** (F). Furthermore, we shall sometimes employ the Algol syntax

if T **then** F
and **if** T **then begin** V; W **end**

as alternatives for

(if T **then** F **else** I)
and **(if** T **then** V; W **else** I)

respectively.

subclass of the computations defined by flowcharts. Secondly, writing **while** programs is often easier and leads to a more compact and elegant program than writing down sets of labelled instructions or drawing flowcharts.

(c) Procedure programs

The final type of control mechanism is found, in one form or another, in every high-level programming language which allows recursive subroutines to be defined. First, we need another set of identifiers, called *procedure identifiers*, which are denoted by $R_1, R_2, \dots$ etc.

A *procedure program* has the form

$$E \textbf{ where } R_1 \textbf{ is } E_1,\ R_2 \textbf{ is } E_2, \dots, R_n \textbf{ is } E_n,$$

where $R_1, R_2, \dots R_n$ are procedure identifiers, and $E, E_1, \dots E_n$ are *expressions*. E is referred to as the *initial* expression, and E_j as the *defining* expression for R_j. Expressions are defined recursively according to the following rules:

1. Each procedure identifier (from the set $\{R_1, R_2, \dots R_n\}$), and each operation identifier, standing by itself, is an expression. So, by convention, is the special null expression, denoted by I.
2 If D and E are expressions, then so are each of the following:

   (2a) $D; E$,

   (2b) (**if** T **then** D **else** E),

   where T is an arbitrary test identifier.

Thus, for example, the following is a procedure program:

$$\begin{aligned}&R_1; R_2 \textbf{ where}\\ &R_1 \textbf{ is } F; (\textbf{if } T \textbf{ then } R_1 \textbf{ else } G; R_2),\\ &R_2 \textbf{ is } (\textbf{if } T \textbf{ then } I \textbf{ else } F; R_1).\end{aligned}$$

Notice the important condition that the procedure identifiers which occur in the expressions $E, E_1, \dots, E_n$ must be among the set $\{R_1, R_2, \dots R_n\}$, i.e. every procedure identifier appearing in the program must have an associated definition.

The computation of a procedure program consists in evaluating the expression E using the given definitions. Every procedure identifier appearing in E is replaced at the appropriate time by its defining expression, and the evaluation continues until E has been transformed into the null expression I. The formal definition of this process is given later.

So far we have defined three types of program which model some of the features of real programming languages in a simple and straightforward manner. However, by themselves, these programs are quite unable to describe computations. Programs need to be supplemented with the *meaning* of the various test and operation identifiers occurring in them. This is exactly what the concept of a machine accomplishes, and this we turn to next.

1.2 Machines

The task of a machine is to supply all the information that is missing from a program in order that computations can be described. First, a machine has to supply the meanings of the operation and test identifiers. Logically, each operation identifier must denote a transformation on the memory structure of the machine, and each test identifier must denote some truth function. In addition, a machine has to describe how to get information into and out of this memory structure, which amounts to providing input and output functions.

Formally, a *machine* M consists of the specification of the following sets and functions:

(a) an *input* set X,

(b) a *memory* set V,

(c) an *output* set Y,

(d) an *input* function $I_M : X \to V$

(e) an *output* function $O_M : V \to Y$,

(f) for each operation identifier F, a function $F_M : V \to V$

(g) for each test identifier T, a function $T_M : V \to \{true, false\}$.

By a function in (d)–(g), we mean a *total* function, i.e. a function that is defined for every element in its domain.

As a simple example, consider the machine M defined by taking

(a) $X = Z$ (the set of all integers, positive and negative).

(b) $V = Z \times Z$,

(c) $Y = Z$

(d) $I_M : Z \to Z \times Z$, defined by $I_M(x) = (x, 0)$

(e) $O_M : Z \times Z \to Z$, defined by $O_M(x, y) = y$,

(f) $F_M : Z \times Z \to Z \times Z$, defined by $F_M(x, y) = (x - 1, y)$
$G_M : Z \times Z \to Z \times Z$, defined by $G_M(x, y) = (x, y + 1)$

(g) $T_M : Z \times Z \to \{true, false\}$ defined by $T_M(x, y) = true$, if $x = 0$
$= false$, otherwise.

When defining machines, we shall be free to choose the operation and test identifiers in such a way as to suggest the nature of their associated functions. For the above example, we can regard the memory set as consisting of two *registers* A and B. Each operation and test acts on one or other of these registers. Moreover, the nature of these operations is captured by the following alternative names for F, G and T:

for F, $A := A - 1$

for G, $B := B + 1$

for T, $A = 0$.