Blackwell Scientific Publications
Computer Science Texts

V.J. Rayward-Smith

A First Course in Computability

COMPUTER SCIENCE TEXTS

A First Course in Computability

V. J. RAYWARD-SMITH
MA, PhD
Senior Lecturer in Computing
University of East Anglia
Norwich NR4 7TJ, UK

BLACKWELL SCIENTIFIC PUBLICATIONS
OXFORD LONDON EDINBURGH
BOSTON PALO ALTO MELBOURNE

Blackwell Scientific Publications
Editorial Offices:
Osney Mead, Oxford, OX2 0EL
8 John Street, London, WC1N 2ES
23 Ainslie Place, Edinburgh, EH3 6AJ
52 Beacon Street, Boston
Massachusetts 02108, USA
667 Lytton Avenue, Palo Alto
California 94301, USA
107 Barry Street, Carlton
Victoria 3053, Australia

First published 1986

Set by Thomson Press (India) Ltd,
New Delhi, and printed and bound in
Great Britain

Distributed in North America by
Computer Science Press Inc., 1803 Research
Blvd, Rockville, Maryland 20850, USA

British Library
Cataloguing in Publication Data

Rayward-Smith, V. J.
The first course in computability.—
(Computer science texts)
1 Electronic data processing
I Title II. Series
001.64 QA76

ISBN 0–632–01307–9

A FIRST COURSE IN COMPUTABILITY

COMPUTER SCIENCE TEXTS

Preface

Much of computer science is centred around the design, analysis and efficient implementation of algorithms to perform various tasks. This book is designed to answer the most fundamental questions that arise from this type of activity—questions such as 'What is a computer capable of?', and given a problem, 'Is there any algorithm to solve it?', and if so, 'Is there an efficient algorithm?' These and similar questions are of such fundamental importance that they must be answered carefully and rigorously.

The rigour in this text is provided by the well-tried discipline of mathematical reasoning. Some computer scientists may find this daunting but there is really little alternative. This text has been written so that a computer scientist in his second or third year should be able to follow all of the arguments. Even a first-year student should reap considerable benefit from reading this material. The concept of solvability is a key idea in computing and the formulation of the concept of unsolvability provides an exciting insight. Once efficiency considerations are introduced the separation of solvable problems into tractable and intractable is a natural next step. The work on intractability has only been developed during the last 15 years but is nevertheless not particularly difficult.

Theoretical computer scientists tend to assume their audience are all familiar with Turing machines. These machines provide a simple computational model which enables us to discover fundamental results. They are widely used and have been widely adopted as the standard model in computability theory. This is why we have used them in this text. Many other equivalent devices would have sufficed; indeed, their use could have simplified some of the presentation. However, tradition in the subject should be respected and so long as the majority of published results are presented in terms of Turing machines, it is important that computer scientists understand their formulation.

Computability is often taught to mathematicians, and rightly so! The use of functions is central to mathematics and a full understanding of them would appear a basic requirement of any honours degree. It must be admitted, however, that many university mathematics departments relegate computability to a final year option. Maybe this book will convince

university teachers that the material can be taught successfully earlier in the course and be understood by their students.

A First Course in Computability is one of three texts which together present the theoretical foundation of computing at an undergraduate level. The other two books in the series are *A First Course in Formal Language Theory* by V. J. Rayward-Smith, and *A First Course in Formal Logic and its Applications in Computer Science* by R. D. Dowsing, V. J. Rayward-Smith and C. D. Walter. All three are published by Blackwell Scientific Publications and it is hoped thereby to provide the academic computing community with a thorough presentation of computing theory. Most theory books are more suitable for advanced undergraduate and postgraduate study. It is hoped that this new series will correct this imbalance and restore theory to its rightful place, central to undergraduate computing degrees.

I would like to express my thanks to my colleagues at the University of East Anglia, Norwich, for their encouragement during the preparation of this text. In particular, I would like to thank Dr G. D. Smith for his comments on earlier drafts. I was particularly fortunate in having an able and co-operative typist, Ms Carol Bracken, without whom the manuscript would have remained an untidy pile of pencilled notes. Finally, I am grateful to Mr Hugh Prior, also of the University of East Anglia, who developed the Pascal program listed in the Appendix as a Turing Simulator.

V. J. Rayward-Smith

Introduction

A naive user of a computer can view it as a 'black box' into which he feeds his data as input and from which he receives output. He does not understand programming; someone else has programmed the machine for him. The computer, as far as this user is concerned, merely takes his input (some string of symbols, x, say) and computes some output. This output, itself a string, generally depends upon x and can thus be regarded as a function of x, $f(x)$. The exact nature of the function, f, will depend upon the program; a different program will compute a different function. (See Fig. 0.1.)

In this book, we will formulate a model for the workings of the 'black box' and its programs. Then we can study all valid programs and the functions they compute. We want our model to be simple enough for easy analysis, yet powerful enough to compute all the functions which, from experience, we expect to be computable—such functions are said to be 'effectively computable'. A formal definition of 'effectively computable' is difficult to obtain because the term is used for a conceptual class of functions, i.e. that class of functions for which a computer scientist would expect to be able to write programs. For example, the function *square*, which takes as input a representation of an integer (say in decimal notation) and delivers a representation of that same integer multiplied by itself, is effectively computable. But to what extent is *square* computable in practice? No real world computer can compute *square*(x) for all integer representations, x. The reason is simple; any real world computer has finite store and thus cannot compute *square*(x) if x is so large that there is not enough store. Yet, we know that given any x, we can compute *square*(x) if only we have a large enough machine. We should not let limits on storage restrict our definition of computability. Thus we can think of an effectively computable function as any function computable on some machine with no limits on

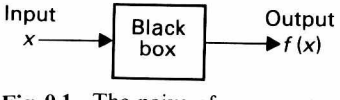

Fig. 0.1. The naive of a computer.

storage. It follows that if our model for the 'black box' is to be able to compute the effectively computable functions, then it must itself be equipped with a limitless store. Then, in any finite computation, only a finite amount of store will be used but there will be no *a priori* limit.

The model for the 'black box' that we will use is that which was proposed by one of the pioneers in this field, Alan Turing (1936). Several alternatives have been proposed, e.g. Post machines (Post, 1936), and unlimited register ideal machines (URIMs) (Shepherdson & Sturgis, 1963). *Church's thesis*, originally hypothesized in 1936 (Church, 1936a) and arising from work by Gödel (1931) and Kleene (1936), can be expressed as the statement that 'every effectively computable function is Turing computable, i.e. is computable on a Turing machine'. Without a rigorous and formal definition of effectively computable, this is an impossible statement to prove formally. However, all the available evidence supports the truth of this hypothesis. Firstly, no one has been able to produce an intuitively computable function that is not Turing computable. Also, other machines designed as a model for a general computing device such as Post machines and URIMs compute precisely the Turing computable functions. Finally, an attempt to produce a generating scheme for all the effectively computable functions defines precisely the Turing computable functions (see Chapter 5). So, whenever one comes across the phrase 'Turing computable', one can infer 'effectively computable', and vice versa.

The Turing machine (TM) is equipped with limitless store which can be viewed as a *tape* divided into a number of locations indexed $\ldots -2, -1, 0, 1, 2, \ldots$ as in Fig. 0.2. In each location, we can write a symbol from some prescribed alphabet. Initially the tape will be blank, i.e. each location will hold the blank symbol, denoted by $\wedge$. The input $x = x_1 x_2 \ldots x_n$ will then be written one symbol at a time in locations $1, 2, \ldots, n$.

The TM has a *head* which moves backwards and forwards across the tape scanning the locations and inserting and deleting symbols. The tape head can be in any one of a finite number of *states*, Q. Initially, the tape

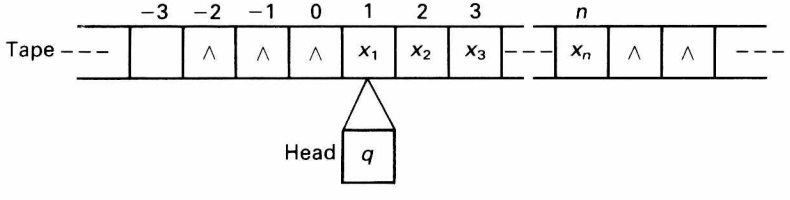

Fig. 0.2. The initial configuration of a Turing machine with input $x = x_1 x_2 \ldots x_n$.

head will be in a prescribed *initial state*, $q_0 \in Q$, and will scan location one. The machine will only halt if the tape head reaches any one of a number of *final states*.

At any time during the computation, the head will then be in some state, q_k, and will be reading some symbol on the tape (called the *current tape symbol*). The next step of the TM depends upon these values. If the machine does not halt, i.e. if q_k is not a final state, the tape head can overwrite the current tape symbol and then possibly move to the adjacent location to the left or to the right. At the same time, the state, q_k, may or may not change. If the machine halts, then its output is taken to be the symbols in locations $1, 2, \ldots, m$ where $m+1$ is the first location greater than 0 which contains a blank symbol.

The precise action to be taken at each step of a computation is defined by the *Turing machine program*. In common with most authors, we will use the phrase 'Turing machine' not only to include the actual machine but also to include a prescribed program. Thus one Turing machine differs from another when its program differs. Any step of a TM program is given as an action depending upon the state of the head and the current tape symbol. These actions are tabulated as a list of 5-tuples of the form given in Fig. 0.3.

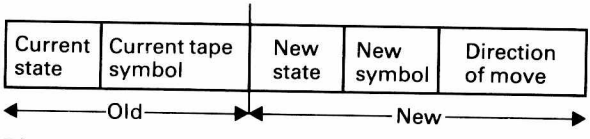

Fig. 0.3.

This list of 5-tuples is the TM program. Hopefully, for any current state which is not a final state and for any current tape symbol likely to arise when the TM is in that state, there will be just one relevant 5-tuple in the list and this will define the new state of the head, the symbol to replace the current tape symbol and the direction in which the head must move (L for left, R for right, 0 for staying put). If a TM program exhibits this uniqueness property, it is said to be *deterministic*.

As an example, let us construct a very simple Turing machine to compute the following. We assume the input is a string, x, comprising a positive number of 1s. Our TM must then result in an output of 1 if the number of 1s in x is even but otherwise it must give an output of 0. Thus this TM computes the *even* function where

$$even(x) = \begin{cases} 1 & \text{if } x \text{ contains an even number of 1s,} \\ 0 & \text{otherwise.} \end{cases}$$

We first describe an algorithm in a 'structured English' code and then we will develop the TM program from this. The essential idea is first to scan the input from left to right and remember whether an even or odd number of 1s has been scanned by using two states: q_0 for even and q_1 for odd. Eventually, we come across a blank symbol. If we are in state q_0, the number of 1s was even and if in state q_1, the number was odd. Thus, we start the TM with our input in locations $1, 2, \ldots$ and in state q_0 scanning location 1. Using $\{\ldots\}$ to enclose comments, the first stage of the algorithm is:

while current-tape-symbol = 1 *do*
    if state = q_0 *then* change to state q_1; move right
    else {state = q_1} change to state q_0; move right
    endif
endwhile;
{current-tape-symbol = $\wedge$} move left

We now want to delete all the 1s on the tape and write the appropriate output in location 1. To achieve this, we now move left deleting all the 1s until we meet a $\wedge$. This must be in location 0. We then move right in preparation for the third stage in which we write the output in location 1. Thus the second stage of the algorithm is:

while current-tape-symbol = 1 *do*
    replace symbol by $\wedge$; move left
endwhile;
    {current-tape-symbol = $\wedge$} move right

The tape head is now at location 1 and we simply write 1 if the current state is q_0 and 0 if the current state is q_1. At the same time, we change state to the halt state, q_2. Hence the final stage of the algorithm is:

{current-tape-symbol = $\wedge$}
if state = q_0 *then* change to state q_2; replace symbol by 1
else {state = q_1} change to state q_2; replace symbol by 0
endif

The first stage of the algorithm can be simply transcribed to the following list of 5-tuples;

$(q_0, 1, q_1, 1, \mathrm{R})$
$(q_1, 1, q_0, 1, \mathrm{R})$
$(q_0, \wedge, q_0, \wedge, \mathrm{L})$
$(q_1, \wedge, q_1, \wedge, \mathrm{L})$

The second stage transcribes to

$(q_0, 1, q_0, \wedge, \mathrm{L})$
$(q_1, 1, q_1, \wedge, \mathrm{L})$
$(q_0, \wedge, q_0, \wedge, \mathrm{R})$
$(q_1, \wedge, q_1, \wedge, \mathrm{R})$

and the third stage to

$(q_0, \wedge, q_2, 1, 0)$
$(q_1, \wedge, q_2, 0, 0)$

We cannot, however, simply list all the 5-tuples of the three stages if we wish our TM to be deterministic and operate as we intend. We overcome this problem by the introduction of new states. We keep q_0, q_1 for the first stage, but use q_0', q_1' for q_0, q_1 in the second stage, and q_0'', q_1'' for q_0, q_1 in the third stage. Our final program is thus

$(q_0, 1, q_1, 1, \mathrm{R})$
$(q_1, 1, q_0, 1, \mathrm{R})$
$(q_0, \wedge, q_0', \wedge, \mathrm{L})$
$(q_1, \wedge, q_1', \wedge, \mathrm{L})$
$(q_0', 1, q_0', \wedge, \mathrm{L})$
$(q_1', 1, q_1', \wedge, \mathrm{L})$
$(q_0', \wedge, q_0'', \wedge, \mathrm{R})$
$(q_1', \wedge, q_1'', \wedge, \mathrm{R})$
$(q_0'', \wedge, q_2, 1, 0)$
$(q_1'', \wedge, q_2, 0, 0)$

where q_0 is the initial state and q_2 is the only final state. We have thus shown that *even* is a Turing computable function. In Fig. 0.4, we illustrate the various configurations of the example TM as it processes as input of $x = 111$.

We can also represent a TM program diagrammatically using a labelled directed graph. We have nodes for each state q in Q. Then, for every 5-tuple (q, a, q', b, X) where q, q' are states, a, b are tape symbols, and X is one of L, R or 0, there is a directed arc from node q to node q' with (a, b, X) as a label (see Fig. 0.5).

Represented as a labelled directed graph, our example TM program is given in Fig. 0.6. The start state is indicated by an inward arrow to the corresponding node, and final states by having their corresponding nodes drawn in square boxes rather than circles. Such representations of TM

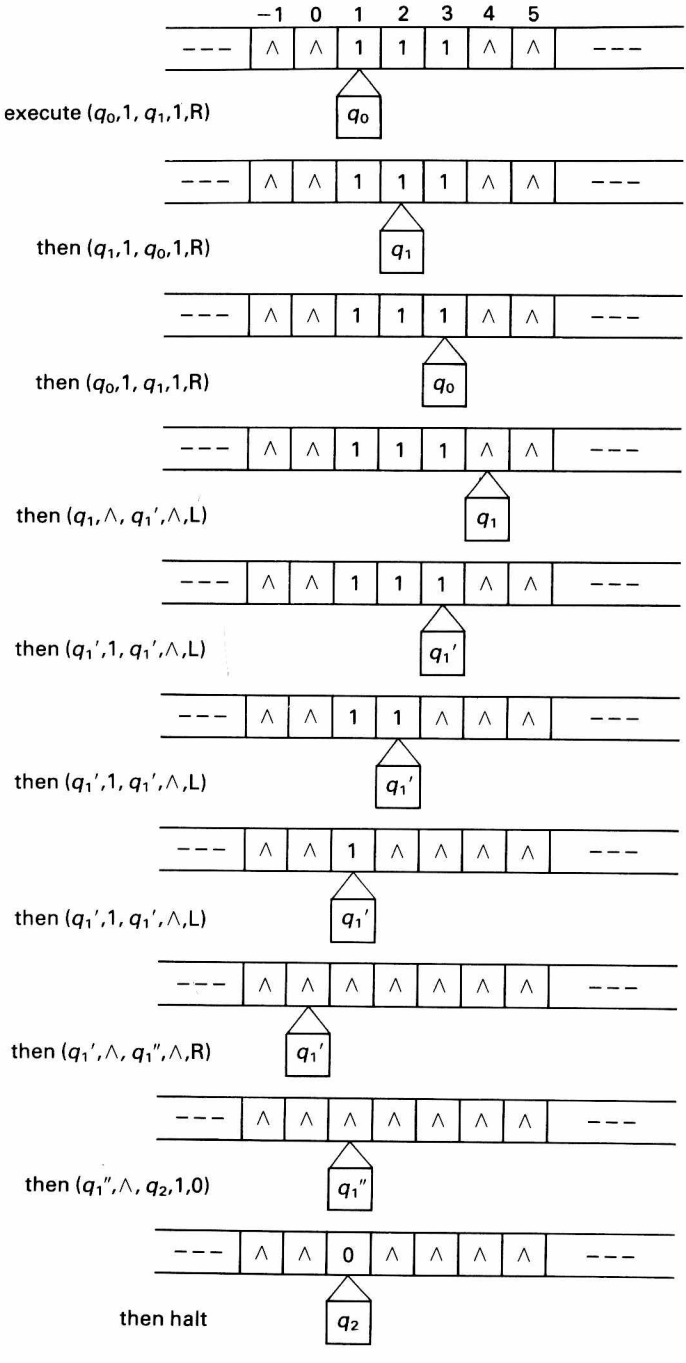
−1 0 1 2 3 4 5
execute $(q_0,1, q_1,1,R)$
then $(q_1,1, q_0,1,R)$
then $(q_0,1, q_1,1,R)$
then $(q_1,\wedge, q_1',\wedge,L)$
then $(q_1',1, q_1',\wedge,L)$
then $(q_1',1, q_1',\wedge,L)$
then $(q_1',\wedge, q_1'',\wedge,R)$
then $(q_1'',\wedge, q_2,1,0)$
then halt

Fig. 0.4.

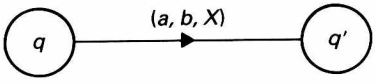

Fig. 0.5.

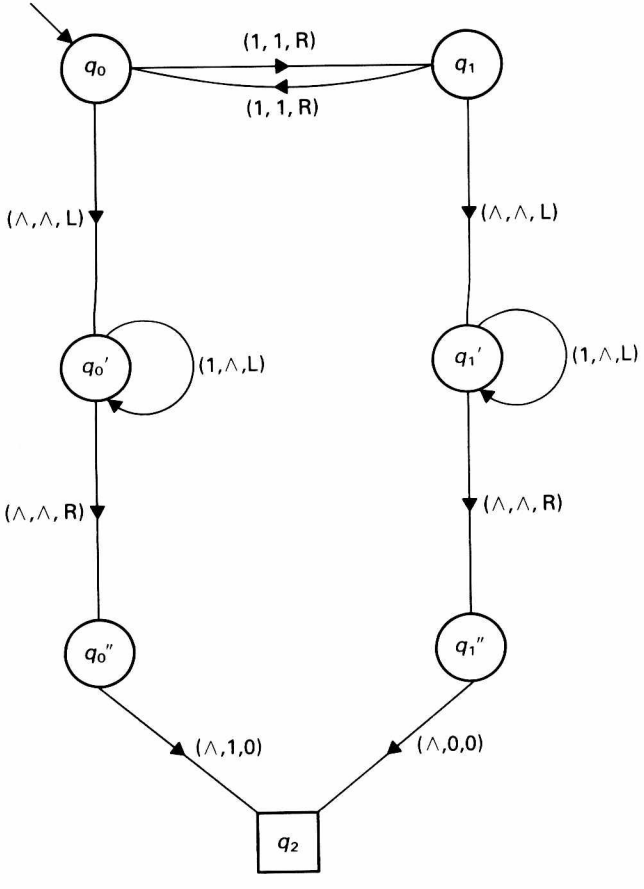

Fig. 0.6.

programs very often provide the greatest clarity. Hence, this will generally be the way that these programs are presented in this book.

We will define Turing machines rigorously in Chapter 2 and, thereafter, investigate their computing power. Firstly, however, we will need to establish our mathematical notation.

Contents

Chapter 1

Mathematical Prerequisites

> But this book cannot be understood unless one first learns to comprehend the language and interpret the characters in which it is written. It is written in the language of mathematics...without which it is humanly impossible to understand a single word of it.
>
> GALILEO GALILEI
> *Il Saggiatore*

In this chapter, we will survey the mathematics required to understand the rest of this book. If this material is new to you, you should study it carefully and make sure you understand all of the concepts. You might like to supplement your reading with Chapters 1, 2 and 5 of the text, *Mathematics for Computing* by McKeown & Rayward-Smith (1982). If, on the other hand, the material is not new to you, this chapter can be skipped through quickly, merely to ascertain the notation that we are going to adopt.

SETS

A *set* is simply a collection of objects without repetition. Each object in a set is called an *element* of that set. If the number of such elements is not too large, then the set can be specified by listing its elements. For example, if D denotes the set of days of the week, then

$$D = \{\text{Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}\}.$$

The elements of a set defined in this way are separated by commas and surrounded by the special brackets { and }. Generally, there is no implied ordering of the elements of a set, so we could equally well have defined

$$D = \{\text{Monday, Wednesday, Friday, Thursday, Sunday, Tuesday, Saturday}\}.$$

If an element, x, is a member of set, A, then we write $x \in A$ [read: x in A] and if x is not an element of A, we write $x \notin A$ [read: x not in A]. Thus,

$$\text{Monday} \in D$$

but Kippers$\notin D$.

Often, however, a set has a large number of elements and perhaps even an infinite number of elements. In such cases the definition of the set cannot be given by listing all its elements and some *defining property* has to be specified. An element, x, is then in the set provided that x satisfies the defining property. A suitable defining property for the set D is

'x is a day of the week'.

So, we could write

$$D = \{x | x \text{ is a day of the week}\}$$

i.e. D consists of all elements, x, that satisfy the defining property. As another example,

$$P = \{x | x \text{ is a prime number}\}$$

defines an infinite set of integers.

When defining a set in this way, care has to be taken to specify the elements under consideration as the *universe (of discourse)*. For example, if

$$X = \{x | x > 2\}$$

then the precise nature of X can only be determined given the values which x might take. For example, if x could only range over the positive integers, X would be a different set from the case where x could range over all real numbers. In the former case $2{\cdot}1 \notin X$ but in the latter $2{\cdot}1 \in X$. Every set under discussion will have all its elements contained in some specified universe, $\mathscr{U}$. Suitable universes in which X might be defined are the integers, the positive reals, etc.

A particularly important set is the *empty set*. This set contains no elements and is denoted by $\varnothing$ or $\{\}$.

We say a set A is a subset of a set B, written $A \subseteq B$, if every element of A is a member of the set B. If A is not a subset of B, we write $A \nsubseteq B$. Thus

$$\{1,2,4\} \subseteq \{1,2,3,4,5\}$$

but $\{2,4,6\} \nsubseteq \{1,2,3,4,5\}$.

It follows from the definition that

$$A \subseteq \mathscr{U}$$

and $\varnothing \subseteq A$ for all sets, A.