The Theory of Computer Science

A Programming Approach

J. M. BRADY

The Theory of Computer Science

A Programming Approach

J. M. BRADY

Department of Computer Science
University of Essex

LONDON

CHAPMAN AND HAI

A Halsted Press Book

John Wiley & Sons, New York

First published 1977
by Chapman and Hall Ltd
11 New Fetter Lane, London EC4P 4EE

Typeset by Alden Press, Oxford,
London and Northampton
and printed in Great Britain by
Richard Clay (The Chaucer Press) Ltd,
Bungay, Suffolk

ISBN 0 412 14930 3 (cased edition)
ISBN 0 412 15040 9 (Science Paperback edition)

Distributed in the U.S.A. by Halsted Press,
a Division of John Wiley & Sons, Inc., New York

Library of Congress Cataloging in Publication Data

Brady, J M
The theory of computer science.

(Science paperbacks)
"A Halsted Press book."
Bibliography: p.
Includes index.
1. Programming (Electronic computers) 2. Machine theory. 3. Computable functions. I. Title.
QA76.B697 001.6'4'01 77-729
ISBN 0-470-99103-8

Preface

The first electronic digital computer was constructed as recently as the 1940s. In the thirty or so years that have elapsed since then, the use of computers has spread to the point where there can now be very few people in Western society whose lives are not affected by computation. The vast majority of companies, government departments, and institutes of tertiary education have their own computer centres, as does a large and growing number of secondary schools.

Although we can hardly begin to conjecture what computers will be used for after another thirty years have gone by, we can safely predict applications such as automated traffic lights, assembly lines, self-piloted vehicles for planetary exploration, automated timetable and route planning services, and machines to work autonomously in environments, such as the ocean bed, which are hostile to man. In fact, prototypes for most of these activities already exist. Equally, the rapidly decreasing cost of computers will mean that, as is already the case with television, a substantial proportion of the population will own one or more computers, whose power will match that of current large machines. We can hardly begin to estimate the impact this will eventually have on the dissemination of information, the quality of life, and the organisation of society.

To date, the development of computers has had an essentially engineering or technological flavour. Spurred on by massive backing from industry and government, new designs and components for machines, and new programming languages, constructs and applications have been developed at an incredible pace. Currently the pace seems not so much to be slowing down as increasing.

Meanwhile, the normal scientific process has been at work. A growing number of workers have sifted and abstracted the many concepts which have been suggested, and have developed models for the analysis of computing phenomena, the explanation of observed regularities, and the suggestion of new lines of development. As the title suggests, this book is an introduction to the theory of using computers. One of the impressions which I hope to convey concerns the vitality, relevance and immediacy of computing theory to computing practice. Chapter 1 explains how the theory is sampled in various ways in the second part of the book in chapters covering the pioneering work of John McCarthy (Chapter 6), attempts to make software reliable (Chapter 7), and ideas regarding the meaning of programming languages (Chapter 8). An introductory text should not aim to be exhaustive in its coverage of the field but should select according to explicitly stated criteria. Two topics which pressed for inclusion in part two, but were eventually excluded, were finite automata and complexity theory. The former was excluded on the grounds that several excellent texts are already

available. The latter was excluded because I felt that current work was more inspired by the ideas of Part 1 than by computing practice.

One interesting anomaly in the development of computer science concerns the fact that the study of precise definitions of 'computable' predated the actual construction of computers! This came about because mathematicians also postulated the notion of 'effective procedure' (what we nowadays call a program) in connection with conjectures about numbers and logic, and sought to make their intuitions precise. Part 1 of the book discusses what we call 'meta' computer science. We shall find that the idea of computability is intimately related to some of the oldest paradoxes of philosophy and the foundations of mathematics. In particular, it is closely related to Gödel's theorem, one of the most remarkable intellectual advances of the twentieth century, which essentially states that there is a limit to what can be proved using formal languages such as second-order predicate logic.

However, the excitement of working in computer science is not of itself sufficient reason for writing a book! There are so many books being produced nowadays that any author ought reasonably to justify adding his contribution. There were, I believe, three main reasons why this book was written, and which distinguish it from most others covering similar material.

Firstly, I strongly believe that the theory of computing must relate directly to a student's intuitions, which are gained from computing practice. These refer to programming, data structuring, machine design, and so on. Thus in Chapter 3 for example we view theorems in recursive function theory as programming problems, and exhibit the surprisingly limited stock of programming techniques which have been developed to date for such proofs. Similarly, Chapter 5 views the problem of proving the equipollence of Turing's formalism and the general recursive functions (GRF) as a computer problem, and thereby exhibits a shortcoming in the GRF.

This approach, especially to the material present in Part 1, should be contrasted with the more usual abstract mathematical treatment, which ignores computing science intuitions, and which results from the historical 'accident' referred to above. One exception to this criticism is Marvin Minsky's (1967) machine-based account. Indeed, this book can be seen as bearing the same relation to Minsky's as programming does to computer hardware design, an observation which finally explains the title of the book.

One notable consequence of taking a programming approach to the theory of computation is that I deny that the main prerequisite for reading the book is a familiarity with modern pure mathematical ideas such as set theory, algebraic structures, and formal logic. Certainly, a familiarity with such ideas would make parts of the material easier to assimilate, but my experience has been that Appendix A contains sufficient mathematical background. The main prerequisite is *computing* experience, and appeals to computing intuitions and practice pervade the book.

The second distinctive feature of the book is closely related to the first. Over

the years I have detected what I believe to be a confusion amongst my fellow computer scientists about the relationship between the theory and metatheory of computer science. I shall argue in Chapter 1 that theory and metatheory are very different ventures, so that to criticise Turing machines, for example, on the grounds that their architecture is not at all like that of von Neumann computers is, quite simply, beside the point.

Finally, I have tried, in Part 2 of the book, to correct the enormous mismatch between what is commonly taught as theory (usually the Part 1 metatheory), and the issues with which computer science theorists concern themselves. Very few computer scientists actively work in recursive function theory. They do work on formal semantics of programming languages, correctness of programs, and so on. At the risk of displaying bias, I have tried to tease out some of the presuppositions underlying models of meaning and correctness, to get the reader to think about their possible shortcomings, and to be aware of the as yet primitive state of computer science.

The approach to the theory of computation outlined in this book developed over five years of teaching an introductory course at the University of Essex. Roughly speaking, it has been used for a two-semester (second year) undergraduate, and a one-semester graduate, course. The course was always accompanied by two projects. The first required the students to implement a Universal Turing Machine (see Checkpoint 2.24) in a suitable high level programming language. The second required students to read about a half dozen research papers on a topic such as lambda calculus models or complexity hierarchies, and to write an essay expressing their understanding and assessment of the material. I have found the latter project particularly valuable for convincing science students, who, as passive consumers, all too often are taught various aspects of Absolute Truth, that controversy, consensus and value judgements are endemic to science.

Finally, I gratefully acknowledge the inspiration, criticism, and encouragement I have received while developing this book. Firstly, some of the workers whose ideas most inspired me to think about the theory of computer science in the first place: John McCarthy, Peter Landin, Rod Burstall, Christopher Strachey, Tony Hoare, John Reynolds, Robin Milner, David Park, David Luckham, Peter Lauer and Bob Floyd. Secondly my colleagues at Essex: Richard Bornat, Pat Hayes, Ray Turner, Bernard Sufrin, John Laski, Tony Brooker, Lockwood Morris, Cliff Lloyd, and Barry Cornelius. Ray Turner, Tony Brooker, and Naomi Brady read various drafts and suggested a great many improvements. Pam Short did a marvellous job of typing. Most of all I acknowledge the support I received from my family.

October, 1976 J. M. Brady

Contents

1 Overview

1.1 Introduction

Computer science has emerged over the past twenty or so years as a discipline in its own right. During that time, a host of new ideas, such as process, data structure, and timesharing, have been established. The primary activity of computing is writing *programs* in *programming languages* which when executed on a *computer* manipulate *data* in such a way as to solve *problems*. The theory of computation chronicles the attempt to establish a theory to account for all of these phenomena in much the same way that classical dynamics attempts to explain the motion of objects.

Any scientific theory is primarily concerned with representing in the abstract realm of mathematics the real-world entities that comprise its subject matter. The representation should facilitate the discovery of mathematical relationships in the form of equations or laws. Thus, in the case of classical dynamics, where the real-world entities include distance, speed, force, and acceleration, distance is usually represented by a function of time, and speed as the derivative of such a distance function. The basic relations are Newton's laws of motion and his laws of gravitation. The relationships discovered within the theory can be interpreted in terms of real-world entities to predict or explain observed real-world phenomena. Alternatively one may be concerned to extend the range of applicability of the theory to accommodate further observations.

Any theory increases our understanding of, and systematizes our knowledge about, the subject domain. But a more immediate reason for constructing theories derives from the fact that it is usually the case that the theory also leads to significant practical advances. So it is with the theory of computer science. Section 1.3, which introduces Part 2 of the book, describes several shortcomings of today's computer science practice in which the theory of computation can be expected to make major contributions.

To each theory there is a corresponding metatheory (Greek: meta = about) which is concerned with analysing the theory itself, for example by precisely defining the concepts used in the theory and in discovering the theory's limitations. Thus, the metatheory of mathematics, naturally called metamathematics, formalizes the notions of set, proof, and theorem, and attempts to discover their limitations. A major result of metamathematics, the undecidability of predicate logic, implies that it is impossible to write a single program that decides for all legal sentences in the logic which are theorems and which are not.

This situation is typical. The theory is concerned with establishing useful *positive* results; the metatheory consists largely of results, often of a *negative*

nature, which define and delimit the subject matter of the theory. Part 1 of the book, introduced in the following section, is an introduction to meta computer science. The major goals are to define precisely what is meant by 'computable' and to establish meta-results such as the unsolvability of the Halting problem for Turing machines, which implies that it is impossible to write a single program P that will decide for any other program Q input to it (in the same way that programs are often submitted as data to a compiler program) whether or not Q will terminate its computation in a finite time. On the other hand, Part 2 of the book will introduce some of the areas in which computer scientists have worked towards the discovery of positive results.

The differing goals of the theory and metatheory of computer science are reflected in the differing approaches to computability which they suggest. Thus the systems we shall meet in Part 1 of the book – Turing machines and the General Recursive Functions – enable results of meta computer science to be proved easily, but are very inconvenient to use. Conversely, systems like McCarthy's which we shall meet in Part 2 (Chapter 6) are much more convenient and practical for proving results *in* but are correspondingly harder to prove theorems *about*. It is unreasonable to indict the formalisms of Part 1 (as many people within computer science do) on the grounds of their impracticality or lack of similarity to modern computers or programming languages; that is simply not their purpose.

As a matter of fact, meta computer science, under its more usual name of Recursive Function Theory, pre-dated the actual construction of computers by several years! For example, Turing machines, which we shall meet in Chapter 2, were introduced in 1936. This apparently paradoxical situation arose largely because (meta)mathematicians had for many years worked on problems which called for 'effective processes' or 'algorithms' (or, as we would now say, 'programs'). In 1900, one of the greatest mathematicians of the century, David Hilbers (1862–1943), delivered a very famous and seminal lecture to the Second International Congress of Mathematicians, in which he posed 23 problems, some of which remain unsolved to this day. The tenth of these problems asked whether it is possible to write a program to decide whether an arbitrary polynomial in a finite number of variables with integral coefficients has integral roots. It was proved by Matyasevich in 1971 that there can be no such program.

One of the distinguishing features of this book is the computer science viewpoint which we adopt, particularly regarding the material in Part 1, which is more usually considered to be a part of mathematics that is somehow relevant to computer science. Instead, we consider the material to be the *metatheory of computer science*. Thus we shall be concerned at all times to relate the ideas we introduce to computer science. We shall rely on intuitions about programming etc., and we will draw our examples from computer science. This is particularly true of Chapter 3, in which we discuss *non*-computable functions or unsolvable problems. The examples to which we give pride of place come from the work of Luckham, Park, and Paterson (1970) and from the theory of formal grammars.

The point about these examples, as opposed to the more usual ones of mathematics or logic (which we also give but not so prominently), is that they were discovered in the course of answering problems posed *in* computer science.

There is one particular idea which we want to promote, that forms the basis of our treatment of the material in Chapters 3, 4, and 5; it can be best introduced via an example. In Chapter 3 we shall be concerned with the question of whether or not various sets S are enumerable. One way to define enumerability is in terms of a computable function $f: N \to S$ from the natural numbers onto S. An alternative viewpoint, less formal but more intuitively appealing to computer scientists, is that a set S is enumerable if there is a program that successively prints the elements of S in such a way that each element of S is listed within a finite time from the start of the process. Accordingly, our 'proofs' of enumerability will usually consist of sketching a program to carry out the enumeration. We thus arrive at the idea of a *program as a constructive proof*. This idea is developed much more fully in Chapters 4 and 5, where we shall show how a programming approach can give real insight into what has traditionally been considered a difficult proof. We will try to convince the reader that he can most usefully approach the Theory of Computation by applying his extensive skill for constructing programs. Hoare and Allison (1972) seem to argue along much the same lines, and we shall refer to their paper several times.

In the final analysis, it will be the richness or poverty of the theory of computation that will determine the scope and standing of computer science. The author will try to convince the reader that the theory is not, as one respected professor of computer science once avowed to him, "as dry as sawdust", but is vital, vibrant, and fascinating.

The remainder of this introductory chapter consists of more detailed overviews of the material to be covered in Parts 1 (Chapters 2–5) and 2 (Chapters 6–8).

1.2 PART 1: Meta computer science

We saw in the preceding section that the two major goals of meta computer science are: (i) to define 'computable' precisely, and (ii) given such a precise definition, to discover what (if any) are the theoretical limitations of computability, thus delimiting the subject matter of computer science.

1.2.1 *Defining 'computable'*

Computer scientists have a deep intuitive grasp of the notion of 'computable' in terms of one or more of 'algorithm', 'effective process', 'program', 'procedure', and 'routine'. However, while we strongly believe that the theory of computation should be presented so as to relate to the intuition of the computer science student, we nevertheless wish to issue a cautionary note against placing too heavy a reliance on intuition at the very basis of the theory of computer science.

In case the experience of set theory and Russell's paradox (see Appendix A, Section A.5.2) does not convince the reader, we have included in the last part of this section a number of comments that expose the limitations of our intuition.

For a computer scientist, the most obvious way to define 'computable is to say that something is computable precisely in the case that it can be programmed in a programming language L, where candidates for L might include FORTRAN, ALGOL 60, and so on. Such a suggestion must be rejected for meta computer science, since, as we saw in the preceeding section, a hallmark of systems in metatheories is that they should be sufficiently *simple* to enable the proofs of metatheorems to be given easily. However, this does not rule out the possibility that the suggestion could profitably be taken up as part of computer science. It turns out that there is a problem with the scheme. We clearly have to be able to deduce for any program exactly what it computes, and this problem – the so-called semantics problem – is far from trivial (see Chapter 8).

As we saw in the preceeding section, since at least 1900, mathematicians and logicians had attempted to give a precise definition of 'computable'. These efforts culminated in 1936 with the simultaneous (independent) publication by Turing in England and Church in the United States of America of suggested definitions which are nowadays accepted as correct.

Church's paper introduced the λ-calculus as a uniform way of denoting computable functions (see Sections 6.1 and 8.2) and he went on to prove the undecidability of predicate logic mentioned above. Nowadays the relevance of Church's work to computer science is seen in its use as the basis of the approach to representing procedures in programming languages as different as LISP (McCarthy *et al.*, 1962) and ALGOL 68 (van Wijngaarden *et al.*, 1969), and its use in establishing a formal definition of the meaning of computing constructs (see Chapter 8).

Turing approached the problem of giving a formal definition of 'computable' directly, by making precise what he argued as his 'intuitive notion'. In a truly remarkable paper he presented an *intuitively-appealing* development of a class of 'abstract machines' which we nowadays call Turing machines (from now on abbreviated to TM). The main point of his paper was to defend the following claim:

> (Turing's thesis) The processes which could naturally be called algorithms are precisely those which can be carried out on Turing machines.

It is important to realize that Turing's thesis cannot be proved or disproved. Rather is embodies a fundamental tenet or law of computer science. The reasonableness or otherwise of this belief can be debated but not proved. (An analogous situation is presented by Newton's 'laws' of motion, which were believed unquestionably until Einstein's theory of relativity suggested a refinement.)

We now present two arguments in favour, and two against, Turing's thesis.

FOR-1 It is difficult to fault Turing's intuitive development of his notion of computable. An alternative intuitive development is presented in Chapter 2.

FOR-2 Since 1936 many great logicians, mathematicians, and computer scientists have framed alternative definitions of 'computable'. All of them have come up with the *same* solution in the sense that the set of computable functions is the same in all cases. The consensus suggests that Turing did indeed discover the correct notion.

AGAINST-1 Turing's system is not rich enough. The fact that Turing machines are so primitive suggests that there are likely to be functions that one would 'naturally' want to accept as computable, but which no Turing machine is capable of computing. (It is probably fair to say that very few people would now advance this argument; however, consider its force in 1936.)

AGAINST-2 Turing's system is far too rich. There are functions that a Turing machine can *in theory* compute but which it is *unrealistic* to accept as computable. For example, it is surely not realistic to call a function 'computable' if any Turing machine to compute it (even if it executes a computational step every microsecond) still would take ten times the life of the universe to calculate an answer.

On balance, Turing's claim is today accepted, despite the misgivings many people have regarding the second argument against it. Indeed, this argument is one of the motivations behind the study of 'complexity' of programs (see Section 1.3).

Broadly speaking, there have been two main approaches to framing a definition of 'computable', which we may call the 'abstract machine approach', and the 'functional' or 'programming' approach.

The former approach, which includes the work of Turing (1936), Wang (1957), Shepherdson and Sturgis (1963), Elgot and Robinson (1964), and Minsky (1967), consists of developing mathematical models of executing mechanisms (processors) on which computations may be 'run'. 'Computable' is then defined to mean 'computable on one of the abstract machines'. The abstract machine approach is illustrated in Chapter 2, which consists of an intuitive development of the Turing machine (TM) model, a discussion of 'Universal' machines, and Shannon's suggested notion of complexity of machines. The chapter ends with a brief look at some other abstract machine approaches.

The functional or programming approach, which includes the work of Church (1941), Herbrand–Gödel–Kleene (see Hermes 1965, Chapter 5), Hilbert–Kleene (the General Recursive Functions), Kleene (1936), and McCarthy (1963), essentially consists of developing mathematical systems of functions for 'programming in'. We shall illustrate this in Chapter 4, by describing the General Recursive Functions and Grzegroczyk's (1953) notion of complexity.

One instance of the second argument (the 'concensus' argument) in support of Turing's thesis is that a function belongs to the General Recursive Functions (GRF) precisely when it is computable according to Turing. Since we are advocating viewing the GRF as a programming language, it is reasonable to ask what the equivalence of the GRF and Turing machines amounts to *in programming terms*. On the one hand we have a linguistic approach GRF, and on the other a

machine approach TM. We have proofs that to each GRF there corresponds a TM, and vice versa. Clearly this suggests a pair of programs: A TM 'simulator' in the GRF, and a 'compiler' from GRF to TM. The main features of such a pair of programs are described in Chapter 5. In particular, it is shown how such a programming approach brings out shortcomings in a system (GRF) apparently intended as part of meta computer science. This view of the equivalence problem as an example of the compiler problem, which we know to be rather difficult, helps us to understand why it was not considered 'obvious' in 1936 that Church and Turing had equivalent notions. (I thank Professor George Barnard for this piece of historical information.)

1.2.2 *The limitations of computability*

Given a precise definition of what it means for a function to be computable, we may ask whether all functions are computable or whether there are some well-defined functions that we can prove are not computable. Put another way, we ask whether there are any well-defined problems that we can prove it is not possible to program a solution to. It is important to realize that by 'possible' we mean 'possible in theory' as opposed to 'financially possible given current resources' or 'possible given the current state of the art of programming'. We devote Chapter 3 to this issue and will see that, as we pointed out in the preceding section, there are indeed problems that we can prove cannot be solved algorithmically. We shall see that the theoretical limitations on what we can program solutions to are extremely subtle; indeed Minsky notes (1967, page vii): "There is no reason to suppose machines have any limitations not shared by man". To get a feel for the limitations of computability, we consider two topics in computer science that give rise to unsolvable problems, namely the equivalence of programs and formal grammars.

1.2.3 *The limitations of our intuition*

To round off this section we present two observations to point up the need for a precise notion of 'computable' on which to build a theory of computer science. Hoare and Allison (1972) present a variant on the same basic theme.

(1) *Descriptions*
An algorithm can be thought of as an unambiguous description of how an executing mechanism is to proceed from step to step. In the final analysis, an algorithm is a description, so it is reasonable to suppose that any algorithm can be described in any powerful enough language, say English. Indeed, many descriptions we find in everyday life can be interpreted as algorithms: Knitting patterns, recipes, instructions from a manual, route-finder maps, and so on. Given that a description is a list of English words, we may order descriptions by saying that a description d_1 precedes description d_2 if either d_1 has less letters than d_2 or, if

they have the same number, d_1 precedes d_2 in dictionary order. Given any set of descriptions we can talk about the shortest (since any totally-ordered set has a least element). Now consider the following description:

> The smallest number whose shortest description requires more than eleven words.

No such number can exist! For if it did, the above *description* would describe it but would take only eleven words! The problem is that we did not impose any restrictions on lists of words that form acceptable descriptions. For more in this direction see Hermes (1965, Chapter 1).

(2) *Self-application* (following Scott, 1970)
Some functions take other functions as arguments; an example is the following general summation function, which is easily programmed in Algol 60:

$$\text{sum}(f, m, n) = f(m) + \ldots + f(n)$$

Another example is the function twice, whose result is also a function:

$$(\text{twice}(f))(x) = f(f(x))$$

so that (twice(sq))(3) = 81. More interesting though is the fact that twice can be applied to itself, for example:

$$\text{twice}(\text{twice})(\text{sq})(2) = 256$$

Unfortunately, we cannot allow functions to be applied to themselves quite generally, since consider the function selfzero defined (cf. Russell's paradox) by:

$$\text{selfzero}(f) = f(f) = 0 \to 1, 0$$

It is all too easy to show that:

$$\text{selfzero}(\text{selfzero}) = 0 \text{ iff } \text{selfzero}(\text{selfzero}) \neq 0$$

The solution to this paradox lies in realizing that selfzero is allowed to take *any* function as an argument. It turns out that there is no problem if selfzero is only applied to computable functions. Selfzero can in fact be programmed in many programming languages; the self-application just does not terminate.

This is not quite as irrelevant as might appear at first glance. Firstly, we would like to be able to program self-applying functions, or at least functions like sum which take *any* computable function as an argument. Secondly, the same problem arises in giving a simplified idealized model of computer storage. Suppose we consider memory to be a set L of locations, each capable of holding any of a set V of values. A state of memory can be modelled by a contents function:

$$\text{cont} : L \to V$$

which gives the value cont (l) contained in the location l. Now one sort of value

that compilers often plant in memory is commands. A command cmd is executed in order to change the state of memory, and so it can be modelled by a state-change function:

$$\text{cmd} : (L \to V) \to (L \to V)$$

Suppose now that cont is a state of memory, and that cont(l) is a command cmd. We can clearly execute cmd in state cont to get a new state cmd(cont). That is, we can execute:

$$\text{cont}(l)(\text{cont})$$

which is just one step away from the self-application problem. Hoare and Allison (1972) also discuss the self-application problem in terms of Grelling's formulation of the Russell paradox. They suggest that one way round the problem is to prevent it from occurring by emulating Russell's set theory solution consisting of a precise theory of types; such a solution is implemented in heavily typed programming languages such as ALGOL 68. Section 8.2 outlines a different, and typeless, solution due to Scott (1970).

1.3 PART 2: Towards a theory of computer science

Part 1 established a framework into which any theory of computation must fit. The *essentially negative* results presented in Chapter 3 *about* the theory show that there is a theoretical limit to what can be computed and to what *positive* results can be achieved *within* the theory. Unfortunately, the metatheory offers hardly any suggestions about the kinds of problem areas that the theory might profitably address, about the kinds of results achievable within those areas, or about the techniques by which those results might be achieved. As we shall see in Chapter 6, even the formalisms of the metatheory are not especially useful when it comes to developing a theory. For example, whereas Part 1 tells us that it is *impossible* to write a single program to determine *for all* programs P and *all* inputs d to P whether or not the computation of P on d terminates, it is nevertheless often *possible* to be convinced about the termination or nontermination for many *individual* pairs P,d; the metatheory remains silent regarding how we might approach the question for a particular pair P,d.

Part 2, consisting of Chapters 6–8, introduces some of the work that has been carried out in the past twenty or so years aimed at filling out the theory with positive results about a number of problem areas. We list below some of the problems that have attracted the most attention to date, and then describe in greater detail the plan of Part 2.

For a theory which is so young, computer science displays a quite remarkable richness, breadth, and vigour, which makes it all the more surprising that there is a widespread opinion that most theoretical studies are esoteric game-playing of no real or lasting value. There seem to be two main reasons for this