高等学校电子信息类专业
“十二五”规划教材

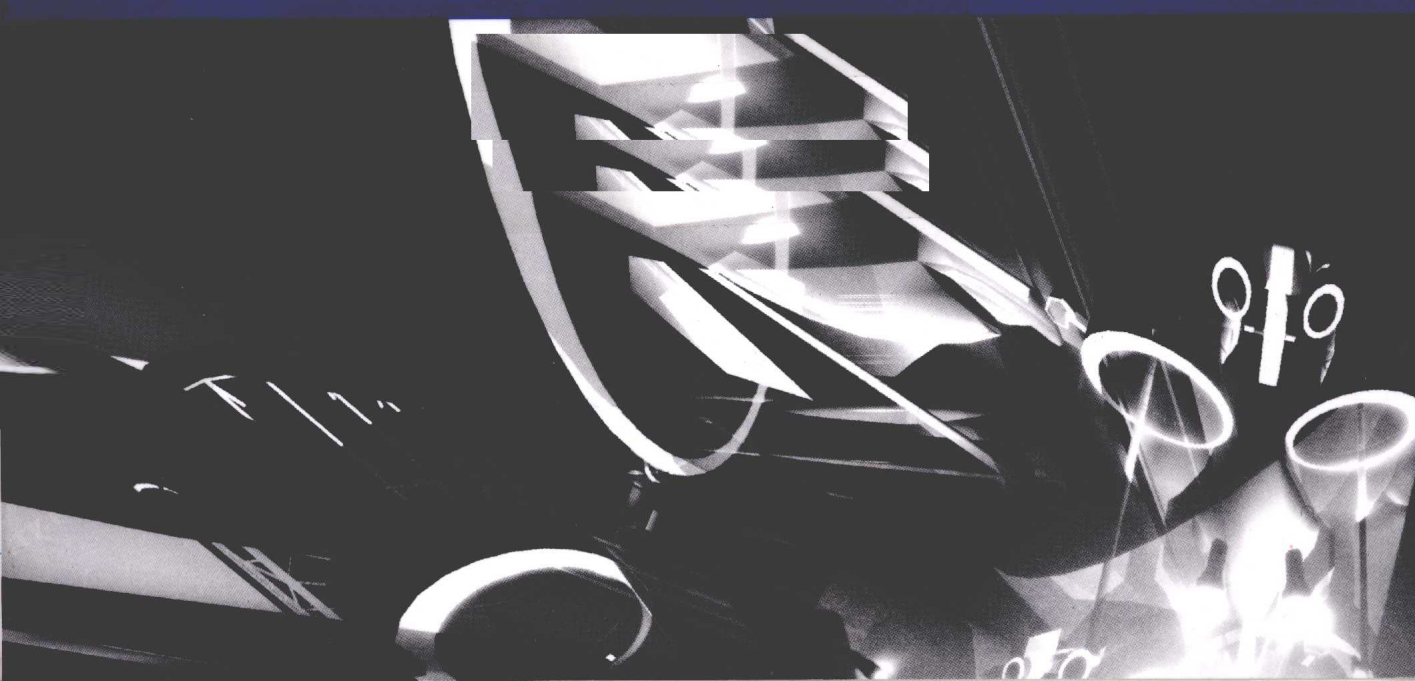
ELECTRONIC
INFORMATION SPECIALTY

信息科学类专业英语

（第二版）

韩俊岗　袁立行　王忠民　编著

西安电子科技大学出版社
http://www.xduph.com



高等学校电子信息类专业“十二五”规划教材

国家级教学团队资助教材

信息科学类专业英语

(第二版)

韩俊岗　袁立行　王忠民　编著

西安电子科技大学出版社

【内 容 简 介】

本书选取了信息科学前沿领域的相关内容，具有内容新、知识面广、难度大、趣味形强等特点。书中主要涉及物联网、云计算、可重构计算、极大规模计算、量子计算、4G无线通信、片上网络、多智能体系统、纳米技术、系统芯片设计、微机电系统、生物信息学、嵌入式系统、可穿戴计算机、人工嗅觉、图形处理器等相关新技术各领域的最新知识，具有一定的广泛性和综合性。在每课末，结合课文内容，均给出了相关的重点、难点句子的解释；同时，为增强学生的阅读兴趣，扩展知识面，提高阅读理解能力，本书采用了大学六级英语考试和研究生入学考试中的阅读理解题的形式，来考查学生对难句、长句和结构复杂句型的理解。

本书既可用作高等理工科院校信息科学与技术，包括电子信息工程、计算机、自动化、微电子学等专业的专业英语教材，又可作为相关领域技术人员的参考书。

图书在版编目(CIP)数据

信息科学类专业英语 / 韩俊岗，袁立行，王忠民编著. —2 版. —西安：西安电子科技大学出版社，2011.2

高等学校电子信息类专业“十二五”规划教材

ISBN 978-7-5606-2546-1

Ⅰ. ①信… Ⅱ. ①韩… ②袁… ③ 王… Ⅲ. ①信息技术—英语—高等学校—教材 Ⅳ. ①H31

中国版本图书馆 CIP 数据核字(2011)第 006648 号

策　　划　云立实

责任编辑　任倍萱　云立实

出版发行　西安电子科技大学出版社(西安市太白南路 2 号)

电　　话　(029)88242885　88201467　　邮　　编　710071

网　　址　www.xduph.com　　电子邮箱　xdupfxb001@163.com

经　　销　新华书店

印刷单位　陕西光大印务有限责任公司

版　　次　2011 年 2 月第 2 版　2011 年 2 月第 3 次印刷

开　　本　787 毫米×1092 毫米　1/16　印　张　14.5

字　　数　338 千字

印　　数　8001～11 000 册

定　　价　22.00 元

ISBN 978-7-5606-2546-1/H · 0135

XDUP 2838002-3

前　言

本书是在2003年出版的《信息科学类专业英语》(西安电子科技大学出版社)的基础上，根据西安邮电学院计算机科学与技术核心课程国家级教学团队教材建设的需要，经过全面修改和补充而完成的，在修订过程中，考虑到近年来信息科学技术的最新发展，补充了超大规模计算、物联网、云计算、4G、生物信息学等热门内容。补充和修改的内容超过了原书的2/3。

多年来的教学经验说明，专业英语对于学生毕业后的就业和进一步深造都很重要。很多通过了四级或六级考试的学生仍不能阅读英语专业文献和技术手册。我们认为信息科学技术领域的学生不能限制在自己的本科专业领域。因为无论是从事信息科学的业务技术开发，还是科学研究，这些领域各个专业的知识和词汇都是交织在一起的。因此，涵盖信息科学与技术的各个主要领域是本书选材和教学的一个基本出发点。另外，专业英语是在学生已经学习了本专业的基础和专业基础课之后开设的，学生渴望了解专业知识的应用和最新发展情况。因此，本书力图选择信息科学各个领域的最新发展成果和趋势。书中的每一课都讲述了一个新领域的主要概念和关键技术，力图给学生展示信息科学的新发展，开拓学生的视野，引发学生的兴趣，为学生的就业和进一步深造提供选择的方向。

本书的主要特点是内容新颖、知识面广、趣味性强。授课对象包括计算机科学与技术、网络工程和电子信息科学与技术、自动化、测控技术与仪器、智能科学与技术等专业的学生。实际上它适用于信息技术领域各个专业的学生，因为我们所选择的文章具有广泛性和综合性。

本书的另一特点是授课难度大，要求授课教师有很广的知识面，且应具有丰富的教学经验。教师可根据实际需要选择部分课程内容，同时应加强专业知识和应用背景的讲解。

多年来，国内的英语教学存在的问题很多。我们认为，学习英语的目的是应用，离开了应用的学习和各种应试的学习都是事倍功半的。

我们希望通过本书的出版和国内教授专业英语的老师互相交流，把我国高校的专业英语教学提高到新的水平，特别希望大家对本书提出宝贵意见。本书出版后，我们可为教师提供参考译文和问题的参考答案，有需要的老师可发E-mail至hjg@xupt.edu.cn。

本书中的英语选文都是从因特网上慎重选择的，我们尽量保持作者的署名，在此向所有原作者致谢。

编　者

2011年1月

目　录

Content

Lesson 1

Towards a Mathematical Science of Computation

J. McCarthy, Stanford University

1 Introduction

In this paper I shall discuss the prospects for a mathematical science of computation. In a mathematical science, It is possible to deduce from the basic assumptions, the important properties of the entities treated by the science. Thus, from Newton's law of gravitation and his laws of motion, one can deduce that the planetary orbits obey Kepler's laws.

What are the entities with which the science of computation deals?

What kinds of facts about these entities would we like to derive?

What are the basic assumptions from which we should start?

What important results have already been obtained?

How can the mathematical science help in the solution of practical problems?

I would like to propose some partial answers to these questions. These partial answers suggest some problems for future work. First, I shall give some very sketchy general answers to the questions. Then, I shall present some recent results on three specific questions. Finally, I shall try to draw some conclusions about practical applications and problems for future work.

2 What are the Entities with Which Computer Science Deals?

These are problems, procedures, data spaces, programs representing procedures in particular programming languages, and computers.

A problem is defined by the criterion which determines whether a proposed solution is accepted. One can understand a problem completely without having any method of solution. Procedures are usually built up from elementary procedures. What these elementary procedures may be, and how more complex procedures are constructed from them, is one of the first topics in computer science. This subject is not hard to understand since there is a precise notion of a computable function to guide us, and computability relative to a given collection of initial functions is easy to define.

Procedures operate on members of certain data spaces and produce members of other data spaces, using in general still other data spaces as intermediates. A number of operations are known for constructing new data spaces from simpler ones, but there is as yet no general theory of representable data spaces comparable to the theory of computable functions.

Programs are symbolic expressions representing procedures. The same procedure may be represented by different programs in different programming languages. We shall discuss the problem of defining a programming language semantically by stating what procedures the programs represent. As for the syntax of programming languages, the rules which allow us to determine whether an expression belongs to the language have been formalized, but the parts of the syntax which relate closely to the semantics have not been so well studied. The problem of translating procedures from one programming language to another has been much studied, and we shall try to give a definition of the correctness of the translation.

Computers are finite automata. From our point of view, a computer is defined by the effect of executing a program with given input on the state of its memory and on its outputs. Computer science must study the various ways elements of data spaces are represented in the memory of the computer and how procedures are represented by computer programs. From this point of view, most of the current work on automata theory is beside the point.

3 What Kinds of Facts about Problems, Procedures, Data Spaces, Programs, and Computers Would We Like to Derive?

Primarily, we would like to be able to prove that given procedures solve given problems. However, proving this may involve proving a whole host of other kinds of statement such as:

1. Two procedures are equivalent, i.e. compute the same function.
2. A number of computable functions satisfy a certain relationship, such as an algebraic identity or a formula of the functional calculus.
3. A certain procedure terminates for certain initial data, or for all initial data.
4. A certain translation procedure correctly translates procedures between one programming language and another.
5. One procedure is more efficient than another equivalent procedure in the sense of taking fewer steps or requiring less memory.
6. A certain transformation of programs preserves the function expressed but increases the efficiency.
7. A certain class of problems is unsolvable by any procedure, or requires procedures of a certain type for its solution.

4 What are the Axioms and Rules of Inference of A Mathematical Science of Computation?

Ideally we would like a mathematical theory in which every true statement about procedures would have a proof, and preferably a proof that is easy to find, not too long, and easy to check. In

1931, Gödel proved a result, one of whose immediate consequences is that there is no complete mathematical theory of computation. Given any mathematical theory of computation there are true statements expressible in it which do not have proofs.[1] Nevertheless, we can hope for a theory which is adequate for practical purposes, like proving that compilers work; the unprovable statements tend to be of a rather sweeping character, such as that the system itself is consistent.

It is almost possible to take over one of the systems of elementary number theory such as that given in Mostowski's book *Sentences Undecidable in Formalized Arithmetic*, since the content of a theory of computation is quite similar. Unfortunately, this and similar systems were designed to make it easy to prove meta-theorems about the system, rather than to prove theorems in the system. As a result, the integers are given such a special role that the proofs of quite easy statements about simple procedures would be extremely long.

Therefore it is necessary to construct a new, though similar, theory in which neither the integers nor any other domain(e.g. strings of symbols) are given a special role. Some partial results in this direction are described in this paper. Namely, an integer-free formalism for describing computations has been developed and shown to be adequate in the cases where it can be compared with other formalisms. Some methods of proof have been developed, but there is still a gap when it comes to methods of proving that a procedure terminates. The theory also requires extension in order to treat the properties of data spaces.

5 What Important Results have been Obtained Relevant to a Mathematical Science of Computation?

In 1936, the notion of a computable function was clarified by Turing, and he showed the existence of universal computers that, with an appropriate program, could compute anything computed by any other computer.[2] All our stored program computers, when provided with unlimited auxiliary storage, are universal in Turing's sense. In some subconscious sense even the sales departments of computer manufacturers are aware of this, and they do not advertise magic instructions that cannot be simulated on competitors machines, but only that their machines are faster, cheaper, have more memory, or are easier to program.

The second major result was the existence of classes of unsolvable problems. This keeps all but the most ignorant of us out of certain Quixotic enterprises such as trying to invent a debugging procedure that can infallibly tell if a program being examined will get into a loop.[3] Later in this paper we shall discuss the relevance of the results of mathematical logic on creative sets to the problem of whether it is possible for a machine to be as intelligent as a human. In my opinion it is very important to build a firmer bridge between logic and recursive function theory on the one side, and the practice of computation on the other.

Much of the work on the theory of finite automata has been motivated by the hope of applying it to computation. I think this hope is mostly in vain because the fact of finiteness is used to show that the automaton will eventually repeat a state. However, anyone who waits for an IBM 7090 to repeat a state, solely because it is a finite automaton is in for a very long wait.

6 How Can a Mathematical Science of Computation Help in the Solution of Practical Problems?

Naturally, the most important applications of a science cannot be foreseen when it is just beginning. However, the following applications can be foreseen.

1. At present, programming languages are constructed in a very unsystematic way. A number of proposed features are invented, and then we argue about whether each feature is worth its cost. A better understanding of the structure of computations and of data spaces will make it easier to see what features are really desirable.

2. It should be possible almost to eliminate debugging. Debugging is the testing of a program on cases one hopes are typical, until it seems to work. This hope is frequently vain. Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs. There is a good prospect of doing this, because we can require the computer to do much more work in checking each step than a human is willing to do. Therefore, the steps can be bigger than with present formal systems.

Vocabulary

1. mathematical science of computation 这里mathematical应当是限定science of computation的，可以翻译成数学(化)的计算科学，意指在计算科学中利用数学方法。
2. sketchy adj. 粗略的。
3. prospect n. 景象，景色，前景，前途视野；展望；眺望；预期；指望 vt. 勘探，勘察，找矿；对……进行仔细调查。
4. deduce vt. 推论，演绎出。
5. semantics n. 语义学。
6. automata n. 自动操作，自动控制。
7. subconscious adj. 下意识的；潜意识的。
8. infallible adj. 不会犯错误的，无过失的；极准确的，极精确的；绝对可靠的；万无一失的；永远有效的。
9. quixotic adj.堂吉柯德式的，空想的，不切实际的。
10. sweeping adj. 包罗万象的，一扫而光的；笼统的，泛泛的，一概而论的；影响广泛的；大范围的。

Important Sentences

[1] In 1931, Gödel proved a result, one of whose immediate consequences is that there is no complete mathematical theory of computation. Given any mathematical theory of computation there are true statements expressible in it which do not have proofs.

在1913年，歌德尔证明了一个结果，这个结果的一个直接推论就是不存在完备的计算数学理论。给定的任何计算数字理论中必定存在可以表达为真的语句，但不存在它的

证明。

[2] In 1936, the notion of a computable function was clarified by Turing, and he showed the existence of universal computers that, with an appropriate program, could compute anything computed by any other computer.

在 1936 年，图灵澄清了可计算函数的概念，并且证明了通用计算机的存在，通过适当的程序，可以做任何其他计算机能够做的计算。

[3] The second major result was the existence of classes of unsolvable problems. This keeps all but the most ignorant of us out of certain Quixotic enterprises such as trying to invent a debugging procedure that can infallibly tell if a program being examined will get into a loop.

第二个主要结果是存在一类不可解的问题。这个结果就使得我们大多数人，除了那些最无知的人之外，不幻想那些唐吉柯德式的事业，例如试图发明能够绝对无误地检验一个程序是否会陷入死循环的查错过程。the most ignorant of us，我们当中最无知的人。

Questions and Answers

(1) According to the text, the entities that Computer Science deals with may not include( ).

A. problems

B. procedures

C. programming languages

D. computers

(2) According to the text, what is one of the first topics in computer science? ( )

A. What these elementary procedures may be?

B. How more complex procedures are constructed from elementary procedures?

C. How to understand a problem completely?

D. What elementary procedures may be, and how more complex procedures are constructed from them?

(3) Which of the following statements describes the relationship between procedures and programming languages. ( )

A. The same procedure may be represented by different programs in different programming languages.

B. The problem of translating procedures from one programming language to another is easy.

C. The problem of defining a programming language semantically can be solved by stating what procedures the programs represent.

D. Computer science must study how procedures are represented by computer programs.

(4) Which of the following statements is wrong for describing the relationship of procedure to data space and programming languages? ( )

A. Procedures operate on members of certain data spaces and produce members of other data spaces.

B. Programs are symbolic expressions representing procedures.

C. The same procedure may be represented by different programs in different programming languages.

D. Programming languages are constructed from different procedure.

(5) According to the text, which of the following statements is wrong about computer?( )

A. Computers are finite automata.

B. A computer is defined by the effect of executing a program with given input on the state of its memory and on its outputs.

C. Much of the work on the theory of finite automata is fruitful when it is applied to computation.

D. Most of the current work on automata theory is beside the point of computer science.

(6) According to the text of section 4, which of the following statement is wrong?( )

A. We would like a mathematical theory in which every true statement about procedures would have an easy short proof.

B. There is no complete mathematical theory of computation.

C. We hope for a theory which is adequate for practical purposes, like proving that compilers work.

D. An integer-free formalism for describing computations has been developed and can be used to prove a procedure terminates.

(7) What important results have been obtained relevant to a mathematical science of computation?( )

A. The notion of a computable function was clarified by Turing.

B. Much of the work on the theory of finite automata has been done.

C. Existence of classes of unsolvable problems.

D. A machine to be as intelligent as a human.

(8) What kind of work does the author suggest not to solve by mathematics science of computation?( )

A. Confirming each feature of a programming language is worth its cost.

B. Proving a program meets its specifications, instead of debugging it.

C. Checking each step of program by human.

D. Better understanding of the structure of computations and of data spaces.

Problems

1. What is the object of computer science?
2. How can mathematics help computer science?

Lesson 2

Extreme Scale Computing

Irving Wladawsky-Berger

Supercomputing has been a major part of my education and career, from the late 1960s when I was doing atomic and molecular calculations as a physics doctorate student at the University of Chicago, to the early 1990s when I was general manager of IBM's SP family of parallel supercomputers.

The performance advances of supercomputers in these past decades have been remarkable. The machines I used as a student in the 1960s probably had a peak performance of a few million calculations per second or megaflops. Gigaflops (billions) peak speeds were achieved in 1985, teraflops (trillions) in 1997, and petaflops (a 1 followed by fifteen zeros) in 2008.

The supercomputing community is now aiming for exascale computing, 1,000,000,000,000,000,000 calculations per second. The pursuit of exascale-class systems was a hot topic at the recent SC09 supercomputing conference.

In the quest for the fastest machines, supercomputers have always been at the leading edge of advances in IT, identifying the key barriers to overcome and experimenting with technologies and architectures that generally then appear in more commercial products a few years later.

Through the 1970s and 1980s, the fastest supercomputers were based on vector architectures and used highly sophisticated technologies and liquid cooling methods to remove the large amounts of heat they generated.

By the late 1980s, these complex and expensive technologies ran out of gas. As the microprocessors used in personal computers and technical workstations were becoming increasingly powerful, you could now build supercomputers using these CMOS micros and parallel architectures at a much lower price than the previous generations of vector machines. A similar transition to microprocessor components and parallel architectures took place in the mainframes used in commercial applications.

Massively parallel architectures, using tens to hundreds of thousands of processors from the PC and Unix markets have dominated supercomputing over the past twenty years. They got us into the terascale and petascale ranges. But, they will not get us to exascale. Another massive technology and architectural transition now looms for supercomputing and the IT industry in general.[1]

Anticipating the major challenges involved in the transition to exascale, the Department Of Energy (DOE) and DARPA launched a series of activities around three years ago to start planning for such systems.

This DARPA *ExaScale Computing Study* provides a very good overview of the key technology challenges. The study identified four major challenges where current trends are insufficient, and disruptive technology breakthroughs are needed to make exascale computing a reality.

The Energy and Power Challenge is pervasive, affecting every part of the system. Today's leading edge petascale systems consume between 2-3 Megawatts (MW) per petaflop. It is generally agreed that an exaflop system must consume around 20 MWs, otherwise their operating costs would be prohibitively expensive. The 1000-fold increase in performance from petascale to exascale must thus be achieved at no more than a 10-fold increase in power consumption.

Such stretch targets were actually achieved in the transition from terascale in the late 1990s to petascale now. But no one believes it can be done again with today's technologies, hence the assumption that a technology and architectural transition as profound as the one two decades ago is now required.

The Memory and Storage Challenge is a major consequence of the power challenge. The currently available main memories (DRAM) and disk drives (HDD) that have dominated computing in the last decade consume way too much power. New technologies are needed.

The Concurrency and Locality Challenge is another consequence of the power challenge. Over the past twenty years we have been able to achieve performance increases through a combination of faster processes and higher levels of parallelism. But, we are no longer able to increase the performance of a single processing element by turning up the clock rate due to power and cooling issues. We now have to rely solely on increased concurrency.

The top terascale systems of ten years ago had roughly 10,000 processing elements. Today's petscale system is up in the low 100,000s. But, because, the only way to now increase performance toward an exascale system is massive parallelism, an exaflop supercomputer might have 100s of millions of processing elements or cores. Such massive parallelism will require major innovations in the architecture, software and applications for exascale systems. This DARPA *Exascale Software Study* provides a good overview of the software breakthroughs required.

Finally, we have the Resiliency Challenge, that is, the ability of a system with such huge number of components to continue operations in the presence of faults. An exascale system must be truly autonomic in nature, constantly aware of its status, and optimizing and adapting itself to rapidly changing conditions, including failures of its individual components. The exascale resiliency challenges are discussed in this DARPA report on *System Resiliency at Extreme Scales*.

There are vast business implications to such a massive technology and architectural transition. For one, the ecosystem of the past twenty years, where PCs have provided the